Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Framework P2P para Execução de Tarefas Bag-of-Tasks com Múltiplas Políticas de Alocação em Ambientes Distribuídos Heterogêneos

Alessandro Ferreira Leite

Brasília
2010

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Framework P2P para Execução de Tarefas Bag-of-Tasks com Múltiplas Políticas de Alocação em Ambientes Distribuídos Heterogêneos

Alessandro Ferreira Leite

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Orientadora
Prof.ª Dr.ª Alba Cristina Magalhães Alves de Melo

Brasília
2010

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Mestrado em Informática

Coordenador: Prof. Dr. Mauricio Ayala Rincón

Banca examinadora composta por:

Prof.ª Dr.ª Alba Cristina Magalhães Alves de Melo (Orientadora) — CIC/UnB
Prof. Dr. Antonio Marinho Pilla Barcellos — UFRGS
Prof. Dr. Ricardo Pezzuol Jacobi — CIC/UnB

CIP — Catalogação Internacional na Publicação

Leite, Alessandro Ferreira.

Framework P2P para Execução de Tarefas Bag-of-Tasks com Múltiplas Políticas de Alocação em Ambientes Distribuídos Heterogêneos / Alessandro Ferreira Leite. Brasília : UnB, 2010.
97 p. : il. ; 29,5 cm.

Dissertação (Mestrado) — Universidade de Brasília, Brasília, 2010.

1. Alocação de Tarefas, 2. Framework P2P, 3. Work Stealing

CDU 004

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Framework P2P para Execução de Tarefas Bag-of-Tasks com Múltiplas Políticas de Alocação em Ambientes Distribuídos Heterogêneos

Alessandro Ferreira Leite

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Prof.ª Dr.ª Alba Cristina Magalhães Alves de Melo (Orientadora)
CIC/UnB

Prof. Dr. Antonio Marinho Pilla Barcellos
UFRGS

Prof. Dr. Ricardo Pezzuol Jacobi
CIC/UnB

Prof. Dr. Mauricio Ayala Rincón
Coordenador do Mestrado em Informática

Brasília, 24 de Junho de 2010

Agradecimentos

Agradeço primeiramente a Deus, que me proporcionou ótimas experiências no decorrer do mestrado. Agradeço aos professores, em especial à minha orientadora, Prof.ª Dr.ª Alba Cristina Magalhães Alves de Melo por aceitar e apoiar o tema dessa dissertação e, sobretudo, pelos seus ensinamentos, paciência e disponibilidade.

Agradeço aos meus amigos: Cícero, Junior, Araribe, Gileno, Flávio, Hammurabi, Éric e a todos que colaboraram direta ou indiretamente para a concretização deste trabalho.

Finalmente, a minha família, em especial aos meus pais, Antero Luiz e Maria de Lourdes por terem sempre me apoiado em tudo.

Resumo

Aplicações em áreas como genômica comparativa, em neurociências ou de controle de tráfego aéreo têm a característica de requerem grande quantidade de processamento de modo a obterem os resultados. Muitas vezes, essas aplicações são executadas em *Desktop grids*, que são uma plataforma baseada em máquinas de uso geral, mas que em grande escala tem potencial de atingirem grande quantidade de processamento. As aplicações em *grid* são em geral aplicações complexas, que executam em um número grande e heterogêneo de nós não dedicados, disponíveis em domínios administrativos distintos. As abordagens atuais de utilização dos recursos em *desktop grids* são centralizadas, possuindo geralmente escalabilidade limitada. O modelo *Peer-to-Peer* (P2P) evoluiu simultaneamente e em paralelo aos interesses da computação em *grid*. Devido à característica descentralizada da arquitetura P2P, ela tem sido utilizada como solução complementar à arquitetura em *grid*. Observa-se, então, uma convergência entre a computação em *grid* e a arquitetura P2P, juntando as melhores características de cada tecnologia. Nesse contexto, a alocação de tarefas é um problema importante que visa atribuir tarefas a um conjunto de recursos, objetivando maximizar o uso dos recursos. A alocação de tarefas em ambientes descentralizados é complexa, uma vez que não há uma visão global do sistema. Nessa dissertação, propomos e avaliamos um framework flexível para alocação descentralizada de tarefas em ambiente de *grid* com múltiplas políticas de alocação. Além do framework, propomos e avaliamos duas políticas de alocação de tarefas baseadas em *Work Stealing*: *Work Stealing with Replication* (WSR) e *Local Work Stealing with Replication* (LWSR). O protótipo do framework proposto foi desenvolvido utilizando JXTA como *middleware* e o Chord como *overlay* P2P. Entretanto, o framework é flexível pois não está acoplado nem ao *middleware* de rede nem ao *overlay* P2P utilizado. Os resultados obtidos em um ambiente heterogêneo composto de 16 máquinas dispostas em dois laboratórios, executando aplicações de controle de tráfego aéreo, mostraram que o tempo de execução da aplicação pode ser sensivelmente reduzido com o nosso framework. Um *speedup* de 11,86 foi obtido para uma aplicação composta de 180 tarefas, reduzindo o tempo de execução seqüencial da melhor máquina de 22,35 minutos para 1,8 minutos, com 16 máquinas.

Palavras-chave: Alocação de Tarefas, Framework P2P, Work Stealing

Abstract

Applications in fields, such as genomics, neuroscience or air traffic control, usually require large amounts of processing power to obtain results. In many times, these applications run on desktop grids, which are a platform based on general-purpose machines that has the potential to reach large amount of processing. Grid applications, in general, are complex applications running on large and heterogeneous non-dedicated nodes, available in different administrative domains. Current approaches to use resource in desktop grids are centralized, having limited scalability. The interest in the Peer-to-Peer (P2P) model grew simultaneously and in parallel to those interests of grid computing. Therefore, we observe nowadays, a convergence between grid computing and P2P computing, aiming to provide the best characteristics of each technology. In this context, task allocation is one important problem where tasks are assigned to a set of resources, aiming to maximize resource usage. Task allocation in a decentralized environment is complex since there is no global view of the system. In this dissertation, we propose and evaluate a flexible framework for decentralized task allocation in a grid environment with multiple allocation policies. Besides the framework, we propose and evaluate allocation tasks based on Work Stealing: Work Stealing with Replication (WSR) and Local Work Stealing with Replication (LWSR). The prototype of the proposed framework was implemented on top of the JXTA middleware using the Chord as overlay. Nevertheless, the proposed framework is flexible because it is decoupled from both the P2P middleware and the P2P overlay. The results obtained in a heterogeneous environment composed of 16 machines arranged in two campus-wide laboratories, executing air traffic control applications show that the execution time can be sensibly reduced with our framework. A speedup of 11.86 was obtained with an application composed of 180 tasks, reducing the sequential execution time of the best machine from 22.35 minutes to 1.8 minutes, with 16 machines.

Keywords: Task Allocation, P2P Framework, Work Stealing

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

Muitas aplicações de áreas como genômica comparativa, neurociências ou controle de tráfego aéreo requerem grande quantidade de poder de processamento e grande número de recursos de modo a obterem os resultados apropriados. Normalmente, os recursos exigidos por essas aplicações podem exceder os recursos disponíveis em um simples computador. Uma maneira de se acelerar a obtenção de resultados é o processamento paralelo. Nesse caso, usa-se mais de um processador com objetivo de executar as aplicações de maneira mais rápida. Tradicionalmente, o processamento é executado por máquinas dedicadas, normalmente homogêneas, que têm o seu uso maximizado pelas aplicações mas com um alto custo. Por conta desse custo, diversos estudos foram realizados no sentido de desenvolver arquiteturas alternativas para propiciar o compartilhamento dos recursos computacionais interligados por uma rede de comunicação, de modo a propiciar a resolução eficiente de problemas ou o compartilhamento dos recursos distribuídos (Foster et al., 2001).

Atualmente, muitas das aplicações que requerem grandes quantidades de processamento são executadas em computadores disponíveis na internet. Com isso, tem-se conseguido poder de processamento na ordem de *Teraflops* com sistemas heterogêneos, que em números absolutos podem chegar na casa das centenas de milhares de máquinas. Esse extremo da computação distribuída é conhecida como *Internet Computing* e, tem possibilitado a execução de aplicações científicas em larga escala a um custo relativamente baixo (Chakravarti et al., 2004).

A plataforma baseada em computadores de uso geral em escala de internet é comumente conhecida como *desktop grids* (Chakravarti et al., 2004). Assim como ocorre nos *grids* computacionais (Grimshaw et al., 1997) (Berman et al., 2003), essa coleção de máquinas distribuídas são agrupadas por uma camada de software, que provê a ilusão de tratar-se de uma única máquina (Abramson et al., 2000) (Taylor et al., 2004).

O termo *grid computing* denota uma infra-estrutura de computação distribuída para aplicações científicas (Foster et al., 2001). Esse nome foi inspirado nas redes elétricas devido a sua pervasividade, facilidade de uso e disponibilidade. As aplicações em *grid* são em geral aplicações complexas, que executam em um número grande e heterogêneo de nós não dedicados, disponíveis em domínios administrativos diferentes.

Ao mesmo tempo, evoluiu em paralelo aos interesses da computação em *grid*, o

modelo de computação *Peer-to-Peer* (P2P). De acordo com a definição mais recente, um sistema P2P pode ser definido como sistema onde os nós se auto-organizam em diferentes topologias, sem necessitar de uma entidade centralizadora (Stephanos and Spinellis, 2004). Devido à característica descentralizada da arquitetura P2P, ela tem sido utilizada como solução concorrente e complementar à arquitetura em *grid*.

Atualmente, observa-se a convergência entre a computação em *grid* e os sistemas P2P (Foster and Iamnitchi, 2003) (Forestiero and Mastroianni, 2009) com objetivo de juntar as melhores características de cada tecnologia. Mais especificamente, muitos ambientes em *grid* estão utilizando técnicas de descentralização P2P para lidar com o compartilhamento de recursos em larga escala. Por outro lado, além das tradicionais aplicações de compartilhamento de arquivos, os sistemas P2P estão sendo utilizados na execução de um conjunto de aplicações científicas que requerem o uso de técnicas de gerenciamento de recursos empregados em sistemas em *grid*.

Dentro desse contexto, o problema de alocação de tarefas em ambientes P2P é relativamente complexo, uma vez que, não há uma visão global do sistema. A alocação de tarefas é um dos mais importantes problemas de sistemas distribuídos. Ela é responsável por mapear um conjunto de tarefas a um conjunto de recursos, objetivando maximizar o uso dos recursos. Este problema é NP-Completo (Graham et al., 1979) e por essa razão, heurísticas são geralmente empregadas para resolvê-lo.

O roubo de tarefas (*work stealing*) é uma política de alocação de tarefas proposta por (Blumofe and Leiserson, 1999) inicialmente para ambientes de memória compartilhada. Em (Jaffar et al., 2004) foi proposta uma versão distribuída que foi executada em um *cluster* homogêneo de 64 máquinas segundo o modelo mestre/escravo. Em (Dinan et al., 2009), uma estratégia escalável de *work stealing* para o modelo de programação PGAS (*Partitioned Global Address Space*) é proposta e testada em um *cluster* de 8192 processadores.

Como as aplicações em *grid* são projetadas para executarem em ambientes geograficamente distribuídos, essas aplicações são geralmente compostas por tarefas independentes e não-comunicantes ditas *Bag-of-Tasks* (BoT). As execuções de tarefas *Bag-of-Tasks* são geralmente aplicações do tipo mestre/escravo. Nesse caso, temos a presença de um nó mestre, que distribui as tarefas aos nós escravos, e ao término da execução, os nós escravos mandam o resultado para o nó mestre. Obviamente, esse modelo mestre/escravo é inadequado para ambientes de larga escala e, por essa razão, técnicas de descentralização P2P têm sido empregadas para resolver esse tipo de problema (Mendes et al., 2009) (Awan et al., 2006). Muitos resultados, até então, são simulações ou então, empregam políticas de alocação simples. Além disso, as pesquisas sobre a tecnologia P2P têm na grande maioria foco em aspectos de baixo nível, de modo que poucos trabalhos têm endereçado as questões referentes ao projeto (*design*) de aplicações P2P, o que tem contribuído para a baixa existência de aplicações P2P em domínios específicos (Walkerdine et al., 2008).

O objetivo da presente dissertação de mestrado é propor e avaliar um framework flexível para alocação descentralizada de tarefas em ambiente de *grid* com múltiplas políticas de alocação. Para avaliar o framework com múltiplas políticas de

alocação, foi implementada uma aplicação real de balanceamento de fluxo de tráfego aéreo.

Problemas de fluxo de rede vem sendo estudados há décadas e, apesar de diversos algoritmos terem sido propostos na literatura, sabe-se que a escolha de um algoritmo particular depende do tipo de problema abordado e do tempo de que se dispõe para resolvê-lo. Dentre os problemas de fluxo de rede, destaca-se o problema de determinação do fluxo máximo, que possui aplicações em diversas áreas. Em particular, o problema tem sido extensivamente estudado na área de Controle de Tráfego Aéreo: re-roteamento de tráfego (Bertsimas and Patterson, 2000), conflitos em fluxos interceptantes (Treleaven and Mao, 2008), divisão equitável de recursos (rotas) em ambientes de Gerenciamento de Fluxo de Tráfego Aéreo (Air Traffic Flow Management - ATFM) distribuídos (Waslander et al., 2006), entre outros.

Para decidir o melhor algoritmo para um determinado cenário usualmente são feitas simulações de diversos algoritmos, o que possibilita escolher aquele que apresente o melhor resultado. Devido ao tamanho das redes de fluxo simuladas, geralmente as simulações são executadas em ambientes distribuídos, onde cada máquina fica responsável pela simulação de um subconjunto de grafos (cenários) em um subconjunto de algoritmos.

Um protótipo do framework proposto nessa dissertação foi implementado e testado com aplicações *BoT* de determinação de fluxo máximo. Para os testes, foram utilizados dados reais de movimentos aéreos dos dias 30 de abril e 02 de maio de 2008 fornecidos pelo Primeiro Centro Integrado de Defesa Aérea e Controle de Tráfego Aéreo (CINDACTA I). Nos testes, as aplicações *Bag-of-Tasks* eram compostas por 90, 180 ou 270 tarefas. Cada tarefa executava um dos algoritmos de fluxo máximo que expressavam os movimentos reais das aeronaves no espaço aéreo. Os grafos utilizados como entrada para os algoritmos de fluxo máximo foram obtidos a partir desses movimentos.

A principal contribuição dessa dissertação de mestrado é o **framework flexível para alocação descentralizada de tarefas em ambiente de *grid***. O framework proposto apresenta uma arquitetura flexível de modo que ele é totalmente desacoplado do *middleware* e do *overlay* P2P. Um protótipo do framework foi implementado utilizando o JXTA (JXTA, 2003) como *middleware* P2P e o Chord (Stoica et al., 2001) como *overlay* P2P. O Chord também foi implementado nessa dissertação. Como contribuição periférica dessa dissertação temos as **políticas de *Work Stealing* com replicação para ambiente descentralizado**. Duas variantes da política de *Work Stealing* foram propostas: *Work Stealing With Replication* (WSR) e *Local Work Stealing With Replication* (LWSR) as quais diferem uma das outras pelo conjunto de nós da rede que podem ser consultados sobre a existência de tarefas. Nas variantes propostas, quando um nó fica ocioso, o nó consulta outros nós por tarefas, a qual é roubada do nó que tenha a sua fila local de tarefas não vazia. Ao contrário da proposta original de *Work Stealing* (Blumofe and Leiserson, 1999), a tarefa roubada não é retirada da fila do nó vítima, mas sim, uma cópia da tarefa é entregue ao nó. Além disso, a escolha da tarefa a ser roubada é realizada de forma aleatória e não do final como ocorre na proposta original.

O presente documento está organizado da seguinte maneira. No capítulo 2, apresenta-se os ambientes P2P e suas principais características. O capítulo 3 apre-

senta as políticas de alocação de tarefas e discute diferentes abordagens e políticas para a alocação de tarefas em ambientes distribuídos. Nesse capítulo, são discutidas as políticas de alocação centralizadas e distribuídas em ambientes heterogêneos. Ao final desse capítulo é apresentado e discutido um quadro comparativo das políticas de alocação de tarefas em ambientes distribuído existentes, destacando as principais características de cada abordagem. O capítulo 4 apresenta o projeto do *framework* para alocação descentralizada de tarefas em ambiente de *grid*, juntamente com as políticas de alocação *Work Stealing* com replicação. Em seguida, o capítulo 5 apresenta e discute os resultados obtidos em nosso ambiente de testes, no qual o framework foi instanciado em uma aplicação de balanceamento de fluxo de tráfego aéreo, utilizando as variantes das políticas de *Work Stealing* propostas para alocação das tarefas. Finalmente, o capítulo 6 apresenta as conclusões do trabalho e indica os possíveis trabalhos futuros a serem desenvolvidos.

Capítulo 2

Ambientes Peer-to-Peer

Este capítulo apresenta algumas definições e características da arquitetura P2P. A primeira seção apresenta as definições mais recentes sobre a arquitetura P2P, juntamente com as características e uso comuns dessa arquitetura. A segunda seção discute a respeito das estruturas dos sistemas P2P e as diferenças entre cada uma. Na terceira seção, são apresentados os principais *overlays* de sistemas P2P existentes. Para cada *overlay*, é apresentado o seu funcionamento geral e características, seguido de um exemplo. Por fim, a quarta seção apresenta alguns ambientes P2P disponíveis na literatura, relacionados com o objetivo dessa dissertação.

2.1 Definição

Não há na literatura um consenso sobre a definição de sistemas P2P. A definição mais tradicional de sistema P2P refere-se a uma arquitetura totalmente distribuída, onde todos os nós são equivalentes em termos de funcionalidade e tarefas que executam (Stephanos and Spinellis, 2004). Esta definição falha ao excluir algumas aplicações aceitas como *peer-to-peer*, como Gnutella (Gnutella, 2000). No entanto, esta definição engloba sistemas que contam com alguns servidores centrais responsáveis por executar um conjunto de sub-tarefas específicas (Stephanos and Spinellis, 2004). Uma outra definição muito aceita é (Shirky, 2000): uma classe de aplicações que tira proveito de recursos como discos e CPU presentes nas bordas da Internet.

Para Stephanos and Spinellis (2004), a razão de não existir um consenso sobre a definição de sistemas P2P está no fato de que os sistemas são ditos *"peer-to-peer"* não pelo que eles executam - suas operações internas - ou arquitetura, mas, como o resultado que produzem são percebidos. Por isso, existem diferentes definições para acomodar casos diferentes de sistemas ou aplicações. As duas características principais de sistemas P2P são (Stephanos and Spinellis, 2004):

- Compartilhamento direto de recursos entre os nodos, sem a intermediação de um servidor centralizado, embora se admita o uso de servidores centralizados na execução de tarefas específicas ou de comunicação.

- Capacidade de auto-organização tolerante a falhas, assumindo conectividade variável e população transiente de nós como norma, automaticamente adaptando-se às falhas nas conexões de rede como em computadores.

Baseado nestas características, sistemas P2P podem ser definidos como (Stephanos and Spinellis, 2004): sistemas distribuídos consistindo de nodos interconectados capazes de se auto-organizar em topologias de rede com o objetivo de compartilhar recursos tais como conteúdo, ciclos de CPU, dispositivos de armazenamento e largura de banda, capazes de se adaptar à população transiente de nodos enquanto mantém conectividade e desempenho aceitável, sem necessitar da intermediação ou suporte de um servidor ou entidade central. Essa será a definição de sistemas P2P adotada nessa dissertação.

Arquiteturas P2P têm sido empregadas em diferentes categorias de aplicações como (Stephanos and Spinellis, 2004):

- Comunicação e Colaboração - são os sistemas que fornecem a infra-estrutura necessária para a comunicação e colaboração, em tempo real, entre os nós, através da voz, mensagens de texto e arquivos, e sem a participação de um servidor central. Como exemplo podemos citar as aplicações de mensagens instantâneas (ICQ, MSN), Skype, Jabber.

- Computação Distribuída - nessa categoria estão os sistemas que têm por objetivo tirar proveito do poder computacional ocioso das entidades participantes. Nesse caso, uma tarefa que demanda tempo de processamento intensivo é quebrada em unidades de trabalhos menores, que são atribuídas aos nodos. Os nodos executam a sua unidade de trabalho e retornam o resultado ao gerador da tarefa. Nesse caso, é usual uma coordenação central responsável por quebrar, distribuir e coletar os resultados. Como exemplo, temos projetos como: Seti@Home (Seti@Home, 2009) e Genome@Home (Genome@Home, 2009).

- Serviço de Suporte à Internet - nessa categoria estão as aplicações criadas para oferecer suporte a serviços de Internet como sistemas *peer-to-peer multicast*, que formam uma infra-estrutura de comunicação que supre a ausência de suporte *multicast* nativo de rede, de forma a permitir que um mesmo conteúdo seja entregue a um número potencialmente grande de nodos.

- Sistemas de Banco de Dados - alguns trabalhos (Stephanos and Spinellis, 2004) têm sido realizados com objetivo de projetar sistemas de banco de dados distribuídos baseados na infra-estrutura P2P.

- Distribuição de Conteúdo - muitas das aplicações P2P estão inseridas nesta categoria, que inclui sistemas destinados ao compartilhamento de arquivos e conteúdo digital entre seus usuários. Os sistemas de distribuição de conteúdo P2P evoluíram de simples sistemas de compartilhamento de arquivos para sistemas mais sofisticados criando um disco distribuído que prover forma de recuperar, indexar, atualizar e recuperar os dados.

Ao contrário dos sistemas centralizados tradicionais, que fornecem acesso a recursos localizados em um único nó, os sistemas P2P fornecem acesso a recursos

localizados em toda uma rede. Isso exige a utilização de algoritmos eficientes para distribuição e recuperação de informação de uma maneira totalmente descentralizada e organizada, equilibrando automaticamente as cargas de armazenamento e processamento de forma dinâmica entre todos os nós de computação participantes, à medida que entram e saem do sistema.

Para Couloris et al. (2007) os sistemas P2P compartilham as seguintes características:

- Seu projeto garante que cada usuário contribua com recursos para o sistema;
- Embora possa haver diferenças nos recursos com que contribuem, todos os nós em um sistema P2P têm as mesmas capacidades e responsabilidades funcionais;
- Seu correto funcionamento não depende da existência de quaisquer sistemas administrados de forma centralizada;
- Os sistemas P2P podem ser projetados de modo a oferecer um grau limitado de anonimato para os provedores e usuários dos recursos;
- Um problema importante para seu funcionamento eficiente é a escolha de um algoritmo para o arranjo dos dados em diferentes nós e o subseqüente acesso a eles, de uma maneira que equilibre a carga de trabalho e garanta a disponibilidade sem adicionar sobrecargas indevidas.

Podem ser identificadas três gerações de desenvolvimento de sistemas e aplicativos P2P (Couloris et al., 2007). A primeira geração foi iniciada com o serviço de troca de músicas Naspter (OpenNap, 2001). Logo em seguida surgiu uma segunda geração de aplicativos de compartilhamento de arquivos, oferecendo uma maior escalabilidade, anonimato e tolerância a falhas, incluindo Freenet (Clarke et al., 2001), Gnutella (Gnutella, 2000), Kazaa (Leibowitz et al., 2003) e BitTorrent (Cohen, 2003) e, por último, a terceira geração, caracterizada pelo aparecimento de *middlewares* para o gerenciamento de recursos distribuídos em uma escala global independente de aplicativos. Como exemplos dessa terceira geração, podemos citar o Pastry (Rowstron and Druschel, 2001), o Tapestry (Zhao et al., 2004), o CAN (Ratnasamy et al., 2001), o Chord (Stoica et al., 2001) e o Kademlia (Maymounkov and Mazieres, 2002).

Essas plataformas são projetadas de modo a colocar os recursos em um conjunto de nós de computação amplamente distribuídos pela rede e para direcionar mensagens a eles em nome dos clientes com garantias do envio de pedidos em um número limitado de passos intermediários, colocando cópias dos recursos em diferentes nós levando em consideração sua disponibilidade, confiabilidade, requisitos de equilíbrio de carga variáveis, localização do armazenamento e uso das informações (Stephanos and Spinellis, 2004).

2.2 Organização

Quanto à organização os sistemas P2P podem ser classificados como estruturados e não estruturados. A organização dos sistemas P2P possui influência significa-

tiva em diversos aspectos, incluindo segurança, robustez e desempenho. Essencialmente, a organização determina as regras para alocação dos objetos ou de suas chaves ao nós, e nos algoritmos de busca empregados.

2.2.1 Sistemas P2P Não Estruturados

Em sistemas P2P não estruturados, a topologia é determinada de forma *ad hoc*, onde as ligações entre os nós são estabelecidas de forma arbitrária, à medida que os nós entram e saem da rede. Nesse caso, a localização dos objetos ou serviços é completamente independente da topologia, o que dificulta a busca dos objetos.

Os mecanismos de busca empregados são geralmente baseados no método de força bruta, com inundação da rede. Nesse caso, é feita a propagação da consulta utilizando busca em largura ou busca em profundidade até o objeto ser encontrado (Stephanos and Spinellis, 2004). Existem também, métodos mais sofisticados e estratégias em termo de uso de recurso como caminhada aleatória (Gkantsidis et al., 2006) ou índices de roteamento (Tsoumakos and Roussopoulos, 2003). Os mecanismos de busca empregados nas redes não estruturadas têm implicações particularmente na disponibilidade, escalabilidade e persistência.

Nesse modelo, o nó que deseja descobrir um recurso envia uma mensagem contendo a consulta aos seus nós vizinhos, juntamente com o Time to Live (TTL) que indica a quantidade de saltos (*hops*) pelas quais a mensagem deve passar antes de ser descartada.

A figura 2.1 ilustra o processo de busca de recurso em um sistema P2P não estruturado. Nesse exemplo, o nó A deseja encontrar um recurso. Para isso, o nó A envia uma mensagem em *broadcast* com TTL igual a três para os nós vizinhos (P e G) que propagam a mensagem para os seus nós vizinhos. Em cada nó, o TTL da mensagem é decrementado em uma unidade até chegar a zero e a mensagem ser descartada. Nesse caso, o recurso estava no nó D, que entra em contato diretamente com o nó A para repassar o recurso. Entretanto, se o recurso estivesse no nó F, a consulta não iria retornar um resultado, apesar do recurso estar presente na rede pois a mensagem seria descartada quando atingisse o nó E. Encontrar um valor de TTL que possibilite encontrar um maior número de recursos sem inundar a rede constitui um desafio para os sistemas que implementam essa estratégia.

Naturalmente, uma desvantagem desse tipo de arquitetura é a falta de escalabilidade, pois à medida que a quantidade de nós aumenta, as mensagens irão atingir apenas uma parte da rede. No entanto, ela é considerada apropriada para acomodar populações de nós com alta rotatividade. Alguns exemplos de sistemas não estruturados são: Publius (Waldman et al., 2000), Naspter (OpenNap, 2001), FreeHaven (Dingledine et al., 2000), Edutella (Nejdl et al., 2002).

2.2.2 Sistemas P2P Estruturados

Os sistemas P2P estruturados surgiram com objetivo de resolver o problema relacionado à escalabilidade dos sistemas não estruturados. Na arquitetura estruturada, a topologia da rede é determinada através de um esquema de alocação de chaves à identificadores dos nós, de modo que os objetos, ou ponteiros para eles, são

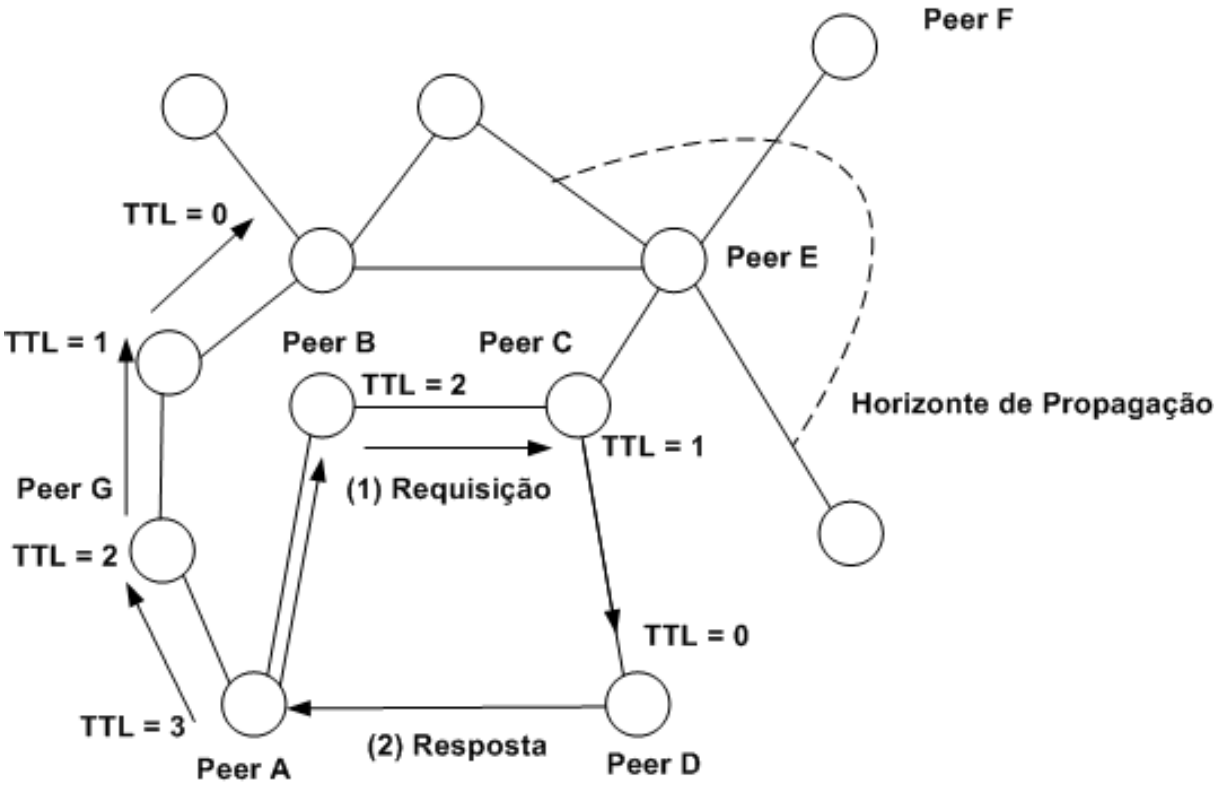


Figura 2.1: Busca em uma rede não estruturada.

colocados em locais calculados deterministicamente. Esses sistemas provêem um mapeamento entre a identificação do objeto e sua localização na forma de tabela de roteamento distribuída (DHT), permitindo que objetos sejam encontrados em um número pequeno de passos.

Dessa forma, os sistemas estruturados oferecem uma solução escalável para consultas exatas, isto é, consultas em que o identificador do objeto é conhecido. Para Barcellos and Gaspary (2007), ao empregar tabelas de roteamento distribuídas é necessária uma correspondência perfeita entre a chave buscada e a chave oferecida como parâmetro.

Uma desvantagem dos sistemas estruturados é que é difícil manter a estrutura requerida para o roteamento eficiente das mensagens, principalmente em redes com populações altamente transientes (Stephanos and Spinellis, 2004).

Exemplos típicos de sistemas estruturados incluem Chord (Stoica et al., 2001), CAN (Ratnasamy et al., 2001), PAST (Druschel and Rowstron, 2001) e Tapestry (Zhao et al., 2004).

Para Stephanos and Spinellis (2004), as categorias de topologia de rede que se situam entre a estruturada e a não estruturada são referenciadas como fracamente estruturadas. Nessa categoria, embora a localização do conteúdo não seja completamente especificada, ela é influenciada por *hits* de roteamento, como por exemplo a Freenet (Clarke et al., 2001). Nesse caso, os nós realizam requisições uns aos outros para armazenar e recuperar objetos.

Segundo Stephanos and Spinellis (2004), os *overlays* estruturados podem ainda ser separados como sendo de infra-estrutura ou de sistemas. *Overlays* de infra-estrutura são infra-estruturas de roteamento escalável em nível de aplicação, como Chord (Stoica et al., 2001), Pastry (Rowstron and Druschel, 2001), Tapestry (Zhao et al., 2004). Por outro lado, *overlays* de sistemas são sistemas completos como OceanStore (Kubiatowicz et al., 2000), PAST (Druschel and Rowstron, 2001) e Kademlia (Maymounkov and Mazieres, 2002).

A tabela 2.1 apresenta as categorias segundo a classificação de Stephanos and Spinellis (2004).

	Centralização		
	Híbrido	**Parcial**	**Nenhuma**
Não estruturado	Naspter, Publius	Kazaa, Morpheus, Gnutella, Edutella	Gnutella, FreeHaven
Infra-estrutura Estruturada			Chord, CAN, Tapestry, Pastry
Sistemas Estruturados			OceanStore, Mnemosyne, Scan, PAST, Kademlia, Tarzan

Tabela 2.1: Classificação dos sistemas P2P quanto à organização (Stephanos and Spinellis, 2004).

2.3 Chord

O Chord (Stoica et al., 2001) é um protocolo *peer-to-peer* de roteamento e localização que realiza o mapeamento de chaves a nós. A localização de dados pode ser implementada utilizando o Chord identificando os objetos com um identificador (chave) e armazenando o par <chave, objeto> no nó responsável pela chave.

No Chord os nós e os objetos são identificados por chaves utilizando uma variação da *hashing* consistente (Karger et al., 1997). Os nós são ordenados em um círculo de 2^m identificadores, onde m é a quantidade de *bits* que formam a chave. Uma chave k é atribuída ao primeiro nó cujo identificador é igual ou maior que k no intervalo de identificação, denominado sucessor de k, denotado por *sucessor(k)*. Se os nós são representados com um círculo numerado de 0 a 2^{m-1}, o *sucessor(k)* é o primeiro nó no sentido horário de k. O uso da *hashing* consistente proporciona uma distribuição uniforme das chaves entre os nós (Stoica et al., 2001).

O identificador do nó ($N.id$) é calculado a partir do seu endereço IP e o identificador da chave ($key.id$) a partir do conteúdo da chave, gerando um identificador de *m-bits* baseado em alguma função de *hash* como SHA-1. Além do balanceamento das chaves, o Chord provê as seguintes funcionalidades (Stoica et al., 2001):

1. **Descentralização**: O Chord é totalmente distribuído e todos os nós têm o mesmo grau de importância, o que torna esta proposta robusta e indicada para aplicações *peer-to-peer*;
2. **Escalabilidade**: Como o custo para localizar uma chave cresce em escala na ordem de $O(logN)$, a localização é factível, mesmo em sistemas com grande quantidade de nós, sem necessitar de nenhum parâmetro de ajuste específico para a escalabilidade;
3. **Disponibilidade**: O Chord conta com um mecanismo que ajusta automaticamente as suas tabelas internas, refletindo tanto as inserções quanto as exclusões de nós. Nesse caso, desconsiderando a existência de problemas nas camadas de rede, o nó responsável por uma chave será sempre localizado.

Para acelerar o processo de localização das chaves, cada nó n mantem uma tabela de roteamento, *finger table*, com no máximo m entradas, sendo m a quantidade de *bits* do identificador da chave/no, informalmente denotada por $n.finger[1, \cdots, m]$, com $n.finger[x], 1 \leq x \leq m$. Observa-se, que os nós mantêm pouca informação sobre os outros nós, mas sobre os nós que encontram-se próximo ao nó no círculo.

Para um nó n realizar uma busca por uma chave k, a *finger table* é consultada para identificar o n' cujo identificador $n.id$ esteja entre n e k. Se esse nó existe, a pesquisa é reiniciada a partir de n', caso contrário, o sucessor de n é retornado.

No Chord, quando um nó n junta-se à rede, algumas chaves atribuídas ao $n's$ sucessores serão re-atribuídas a n. Da mesma maneira, quando um nó n desconecta da rede, todas as suas chaves são re-atribuídas ao seus sucessores. Essas são as únicas mudanças necessárias na atribuição das chaves para manter o balanceamento.

Na figura 2.2 temos um exemplo da estrutura do Chord com 10 nós e m = 6. Para exemplificar o processo de pesquisa pela chave $k = 54$, iniciando a consulta a partir do nó 8 sem o uso da *finger table*, seriam necessários 8 redirecionamentos ($N14$, $N21$, $N32$, $N38$, $N42$, $N51$, $N56$) para encontrar o nó ($N56$) responsável pela chave. No entanto, ao utilizarmos a *finger table* serão necessários 3 redirecionamentos para que o nó ($N56$) que é o sucessor de k seja encontrado, conforme representado na figura 2.3. Nesse caso, primeiro o nó verifica se a chave k é igual ao seu identificador. Se forem iguais, o nó é retornado, caso contrário, o nó verifica se k está entre ele e o seu sucessor. No exemplo, como a chave $k = 54$ não está associada ao nó $N8$ nem ao seu sucessor $N14$ a *finger table* é consultada e a pesquisa reinicializa a partir do nó $N42$, que é o maior identificador da *finger table* que não é maior que k.

A inserção de nó pode afetar a pesquisa de alguma chave k que tenha sido solicitada antes da estabilização do círculo. Nesse caso, três situações distintas podem ocorrer. Na primeira situação, as *finger tables* mantidas pelos nós envolvidos na pesquisa encontram-se atualizadas, possibilitando a identificação do *sucessor(k)* em

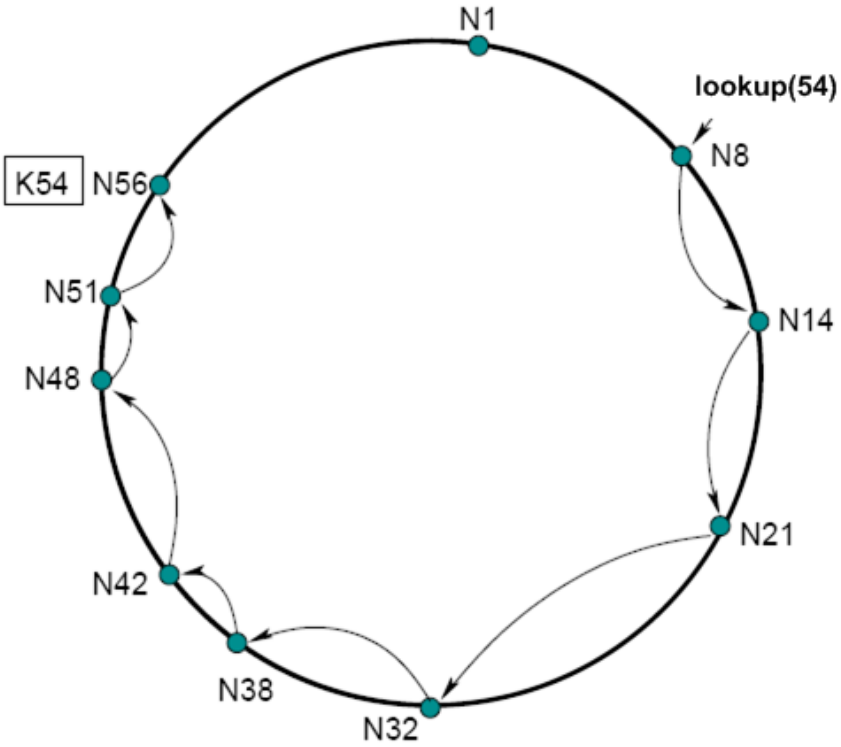


Figura 2.2: Representação da pesquisa da chave $k = 54$ em um sistema com 10 nós, $m = 6$, sem utilizar a *finger table* (Stoica et al., 2003).

$O(logN)$. Na segunda situação, os *links* para os sucessores encontram-se corretos, no entanto as *finger tables* contêm informações imprecisas. Nesse caso, a pesquisa será lenta. Na terceira situação, os nós da região afetada pela inserção apresentam *links* para os sucessores incorretos ou as chaves ainda não foram migradas para os novos nós inseridos. Conseqüentemente, a pesquisa irá falhar. Cabe, então, a camada superior solicitar uma nova pesquisa após um intervalo de tempo (Stoica et al., 2001).

Segundo (Stoica et al., 2001), o modelo utilizado pelo Chord garante que a inserção de novos nós preserva a localização dos nós existentes, mesmo em função de reordenações, perdas ou inserções de nós de forma concorrente. Problemas com o surgimento de topologias com múltiplos círculos desconexos ou círculos em *loop* não podem ser ocasionados pelas ações de inserção, remoção ou re-ordenamento. Se um nó n falhar, os nós cujas tabelas incluem n devem localizar o sucessor de n. Além disso, a falha do nó n não deve causar a interrupção da pesquisa em andamento e, enquanto isso o sistema se estabiliza novamente. Um dos exemplos de uso do Chord é o *Cooperative File Systems* (CFS) (Dabek et al., 2001). Além disso, o Chord tem sido utilizado como ferramenta para servidor de DNS (Cox et al., 2002) e como banco de dados distribuído de chaves públicas (Ajmani et al., 2002).

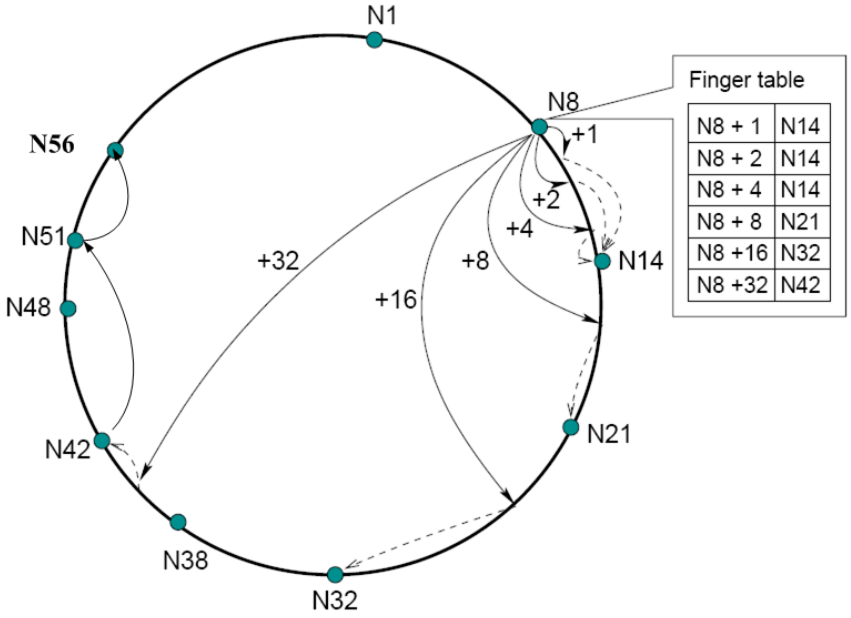


Figura 2.3: Representação da pesquisa da chave $k = 54$ em um sistema com 10 nós, $m = 6$, utilizando a *finger table* (Stoica et al., 2003).

2.4 Tapestry

O Tapestry (Zhao et al., 2004) é uma infra-estrutura *peer-to-peer* que permite a localização de objetos e o roteamento de mensagens a objetos de forma distribuída, auto-administrada e tolerante a falhas, proporcionando estabilidade ao sistema, ignorando rotas e nós sem conectividade e adaptando rapidamente a comunicação da topologia a essas circunstâncias (Zhao et al., 2004).

A topologia da rede se auto-organiza à medida que os nós entram e saem da rede. As informações de localização e roteamento são distribuídas entre os nós da rede. A consistência da topologia é verificada dinamicamente e em caso de perda ou falhas, ela é reconstruída ou atualizada, através da utilização de algoritmos adaptativos.

O Tapestry é baseada nos mecanismos de localização e roteamento introduzidos por (Plaxton et al., 1997), conhecidos como malha Plaxton (*Plaxton mesh*). A malha Plaxton é uma estrutura de dados distribuída que permite aos nós localizarem objetos e rotearem mensagens em uma rede de tamanho arbitrário, usando mapas de roteamento pequenos e de tamanho constante. Na malha Plaxton original, os nós podem assumir o papel de servidores (onde os objetos são armazenados), roteadores (que encaminham mensagens) e clientes (que originam mensagens).

No Tapestry, cada nó mantém um mapa de vizinhos. O mapa de vizinhos possui múltiplos níveis, onde cada nível n contem ponteiros para nós cujos identificadores (ID) devem coincidir (*matches*) com n dígitos. Por exemplo: $4 * ** \Rightarrow 42 * * \Rightarrow 42A*$ $\Rightarrow$ 42AD, onde os *'s representam curingas. Cada entrada na tabela de vizinhos corresponde a um ponteiro para o nó mais próximo na rede cujo ID coincide com o número no mapa.

Na tabela 2.2, temos um exemplo da tabela de mapas de vizinhos mantida por

	Nível 5	Nível 4	Nível 3	Nível 2	Nível 1
Entrada 0	07493	*0493	**093	***03	****0
Entrada 1	17493	*1493	**193	***13	****1
Entrada 2	27493	*2493	**293	***23	****2
Entrada 3	37493	*3493	**393	***33	****3
Entrada 4	47493	*4493	**493	***43	****4
Entrada 5	57493	*5493	**593	***53	****5
Entrada 6	**67493**	*6493	**693	***63	****6
Entrada 7	77493	*7493	**793	***73	****7
Entrada 8	87493	*8493	**893	***83	****8
Entrada 9	97493	*9493	**993	***93	****9

Tabela 2.2: Tabela de vizinhos do nó com ID 67493.

um nó com ID 67493. A quinta entrada do nível $n = 3$ para o nó 67493 aponta para o nó mais próximo a 67493, cujo ID termina com **593.

Portanto, as mensagens são roteadas incrementalmente para o nó destino, dígito a dígito da direita para a esquerda. A figura 2.4 apresenta um exemplo do caminho tomado por uma mensagem do nó 67493 para o nó 34567. Os dígitos são resolvidos da direita para a esquerda da seguinte forma: ****7 $\rightarrow$ ***67 $\rightarrow$ **567 $\rightarrow$ *4567 $\rightarrow$ 34567.

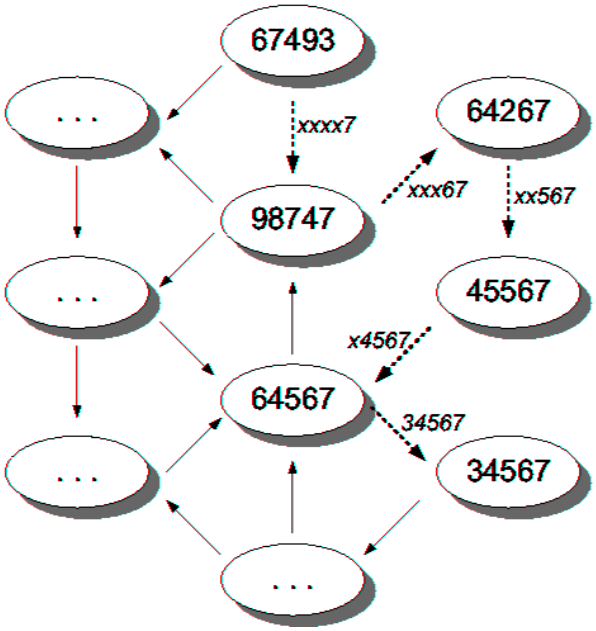


Figura 2.4: Caminho tomando pela mensagem do nó 67493 para o nó 34567 (Stephanos and Spinellis, 2004).

A malha Plaxton usa um nó raiz para cada objeto, que serve como uma garantia a partir do qual um objeto pode ser localizado. Quando um objeto o é inserido na rede e armazenado no nó n_s, um nó raiz n_r é associado ao objeto usando um algoritmo determinístico global (Zhao et al., 2004). Uma mensagem é então roteada

de n_s para n_r, armazenando dados na forma do mapeamento (id do objeto *o*, id do nó que armazena n_s) para todos os nós existentes ao longo do caminho. Durante uma operação de busca, mensagens destinadas para *o* são inicialmente roteadas em direção a n_r, até que um nodo seja encontrado contendo o mapeamento (*o*, n_s).

O Tapestry estende a malha de Plaxton (Plaxton et al., 1997), de modo a aceitar populações transientes de nós, pois Plaxton possui a limitação de ter um conhecimento global para designar e identificar os nós raízes e com isso, assume uma população estática de nós. Além disso, o Tapestry possui tolerância a falha adaptativa. O Tapestry é utilizado por vários sistemas como OceanStore (Kubiatowicz et al., 2000), Mnemosyne (Hand and Roscoe, 2002) e Scan (Chen et al., 2002).

2.5 Skip Graphs

Skip Graph (Aspnes and Shah, 2003) é uma estrutura de dados distribuída baseada em *Skip list* (Pugh, 1990), que provê as funcionalidades de árvore balanceada em um sistema distribuído, onde os objetos são armazenados em nós que podem apresentar falhas a qualquer momento, projetada para ser utilizada em operações de busca em uma rede P2P, possibilitando a consulta baseada em chaves ordenadas (Aspnes and Shah, 2003) (Aspnes and Shah, 2007).

A *Skip list* é uma estrutura de árvore balanceada organizada em níveis de listas encadeadas esparsas, como representado na figura 2.5. O nível 0 em uma *Skip list* é uma lista encadeada de nós ordenados de forma crescente. Para cada $i > 0$, cada nó no nível $i - 1$ aparece no nível i com uma mesma probabilidade p. Em uma *Skip list* duplamente encadeada, cada nó armazena um ponteiro para o predecessor e um para o sucessor para cada lista com uma média de $\frac{2}{1-p}$ ponteiros para cada nó.

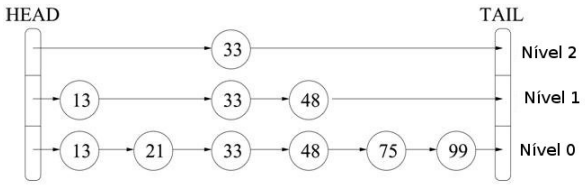


Figura 2.5: Skip list com 6 nós e 3 níveis (Aspnes and Shah, 2007).

Em *Skip Graph*, assim como em *Skip list*, cada nó é membro de múltiplas listas encadeadas, com a lista do nível 0 contendo todos os nós ordenados em seqüência. O que diferencia um *Skip Graph* de uma *Skip list* é que há muito mais listas em um nível i, e cada nó participa de uma dessas listas até os nós serem divididos em *singletons*, após $O(logn)$ níveis na média. Um *Skip Graph* suporta as mesmas operações de inserção, exclusão e busca, análogo às operações de uma *Skip list*.

Formalmente, um *Skip Graph* pode ser definido da seguinte forma. Seja $\sum$ um alfabeto finito, e $\sum^*$ o conjunto de todas as palavras formadas pelos caracteres de $\sum$, $\sum^w$ o conjunto de todas as palavras infinitas e $|w|$ o tamanho de uma palavra, com $w = \infty$, se $w \in \sum^*$. Uma palavra w é formada por $w_o w_1 w_2 \cdots$. Se $|w| \geq i$,

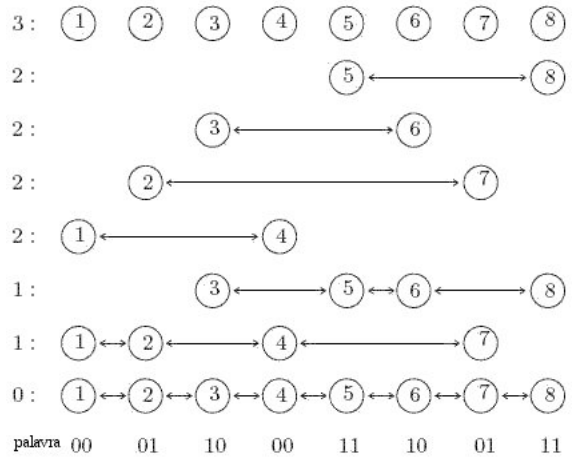


Figura 2.6: *Skip Graph* com 8 nós e 3 níveis (Mendes et al., 2009).

o prefixo de w de tamanho i é denotado por $i = w[0 \cdots i-1]$. Uma palavra vazia, $|w| = 0$, é representada por ϵ. Em um *Skip Graph* o nível 0 é sempre uma lista duplamente encadeada denominada S_ϵ, com todos os elementos em ordem. Em geral, as listas ligadas de nível i são denominada S_w para todo $w \in \sum^*$.

Um *Skip Graph* também pode ser pensado com um grafo randômico, onde existe uma aresta entre x e y quando x e y são adjacentes em alguma S_w (Aspnes and Shah, 2007).

Em um sistema *peer-to-peer*, cada recurso será um nó em um *Skip Graph* e os nós são ordenados de acordo com a chave de cada recurso. Cada nó armazena o endereço e as chaves dos nós adjacentes de cada $O(logn)$ níveis. Dessa forma, cada nó também necessita de $O(logn)$ de espaço para armazenar o seu vetor de membros.

A operação de busca em *Skip Graphs* é realizada da seguinte forma: iniciando a partir de qualquer nó, o algoritmo percorre as listas encadeadas nas direções esquerda ou direita (L ou R) utilizando os níveis mais altos como caminhos (*"express lanes"*) para os níveis mais baixo. Suponha que desejamos realizar uma busca pelo nó 1 a partir do nó 8. Nesse caso, como no nível 3 temos oito listas de um elemento, teremos que a partir do 8 visitar o nó 5 no nível 2, por sua vez, o nó 5 visita o nó 3 no nível 1 e finalmente, o nó 3 visita o nó 2 no nível 0 da lista que encontra o nó 1 a sua esquerda, conforme mostra a figura 2.7.

2.6 Ambientes para Aplicações P2P

As pesquisas sobre a tecnologia P2P têm na grande maioria foco em aspectos de baixo nível, de modo que poucos trabalhos têm endereçado as questões referentes ao projeto (*design*) de aplicações P2P (Foster and Iamnitchi, 2003). Isso tem contribuído para a baixa existência de aplicações P2P em domínios específicos (Walkerdine et al., 2008).

Apesar de existirem implementações de protocolos de redes P2P e APIs (*Application Programming Interface*) para auxiliar o desenvolvimento de aplicações, a complexidade dessas APIs varia significativamente, de simples funcionalidade de

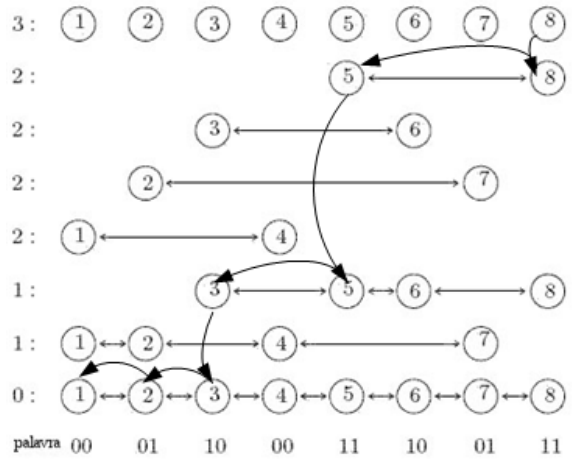


Figura 2.7: Exemplo de busca em um *Skip Graph* com 8 nós e 3 níveis pelo nó 1 a partir do nó 8.

rede provida pela API do Gnutella (Gnutella, 2000), como JTella (JTella, 2009) e pela API do Chord (Stoica et al., 2001) como Accord (Accord, 2009) à funcionalidades mais complexas e orientadas a aplicações como JXTA (JXTA, 2003) e Groove (Groove, 2000). Cada uma dessas APIs requerem do desenvolvedor conhecimento detalhado da tecnologia P2P. Para (Foster and Iamnitchi, 2003), a ausência de padronização da comunidade P2P é visível através da análise da estrutura dessas APIs, que são difíceis de estender e são raros os casos em que uma API tenha sido combinada com outra.

Alguns trabalhos têm sido realizados com objetivo de fornecer uma abstração das tecnologias P2P e de diminuir a ausência de padronização dessas tecnologias, fornecendo uma interface comum aos desenvolvedores e ocultando os detalhes inerentes à heterogeneidade da tecnologia P2P. A *Common API for Structured P2P System* (Dabek et al., 2003), PROST (Portmann et al., 2004), o projeto *Open Overlay* (Open Overlay Project, 2004), o OurGrid (Cirne et al., 2006), Framework for P2P Application Development (Walkerdine et al., 2008) e o P2P Agent Platform for Ubiquitous Service (PIAX) (Fujiwara et al., 2008) são algumas dessas APIs encontradas na literatura. Dentre essas APIs, destaca-se o OurGrid (Cirne et al., 2006), o Framework for P2P Application Development (Walkerdine et al., 2008) e o P2P Agent Platform for Ubiquitous Service (PIAX) (Fujiwara et al., 2008) por fornecerem um framework P2P completo, os quais serão descritos nas seções subseqüentes.

Um framework geralmente é definido como um *design* de larga escala que descreve como um programa é decomposto em um conjunto de objetos que interagem, representados por um conjunto de classes abstratas e da forma como as suas instâncias interagem (Johnson, 1997; Wirfs-Brock and Johnson, 1990).

Um framework de aplicações P2P é um mecanismo para ajudar o desenvolvedor a construir aplicações P2P, fornecendo camadas de abstração que ocultem do desenvolvedor, a complexidade inerente a tecnologia P2P (Walkerdine et al., 2008). A conseqüência dessa abstração é que o desenvolvedor, para instanciar o framework, não necessita conhecer as especificidades da tecnologia P2P, ou seja, não neces-

sita conhecer como a tecnologia funciona, ficando livre para concentrar esforços na construção da aplicação, que será visualizada pelo framework como um *plugin* que utiliza-o. Além disso, essa abstração favorece a independência da tecnologia P2P utilizada, proporcionando que diferentes implementações do framework para uma funcionalidade específica seja utilizada, sem que seja necessária modificações no *plugin*.

Para auxiliar o desenvolvedor na construção das aplicações, o framework deve fornecer um conjunto de funcionalidades acessíveis genéricas, como serviços. Os serviços básicos que devem ser fornecidos, por estarem presentes em um conjunto de aplicações P2P, são: comunicação e compartilhamento de recursos, incluindo a sua localização. No contexto de aplicações de submissão de tarefas, as funcionalidades comuns às aplicações desse domínio incluem a submissão das tarefas, consulta a situação de execução das tarefas, muitas vezes dependentes da tecnologia de rede, e respostas ao solicitante após a execução de uma tarefa ter sido concluída, mas, mantendo uma independência da tecnologia ou API utilizada na implementação.

2.6.1 OurGrid (Cirne et al., 2006)

O OurGrid (Cirne et al., 2006) é um *free-to-join*, *peer-to-peer middleware grid*, ou seja, um *middleware* que utiliza uma abordagem P2P que favorece a criação de um *grid* multi-organizacional para execução de aplicações *bag-of-tasks* (Cirne et al., 2006). O OurGrid adota um modelo hierárquico em que as redes locais, cada qual composta de máquinas que pertencem a uma única organização, são ligadas em conjunto, utilizando uma camada P2P. A figura 2.8 apresenta a arquitetura do OurGrid.

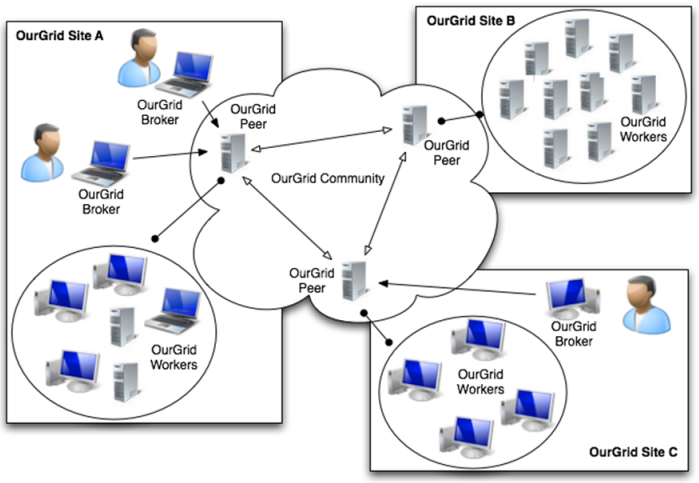


Figura 2.8: Arquitetura do OurGrid Cirne et al.(2006).

Como mostra a figura 2.8, o OurGrid possui três componentes principais: *OurGrid Worker*, *OurGrid Resource Manager Peer*, ou simplesmente *OurGrid Peer* e o *OurGrid Broker*.

O *OurGrid Worker* é um agente que executa nas máquinas da rede e é responsável por implementar um ambiente seguro para a execução de tarefas das aplicações

do *grid*. A política é definida pelo proprietário do recurso, segundo o comportamento oportunidade do *OurGrid*. O *OurGrid Worker* executa um algoritmo responsável por detectar a ociosidade do nó e com isso disponibilizá-lo como um recurso do *grid*, e interrompe a execução de qualquer tarefa no nó quando, considerando a política local, o nó não está mais ocioso.

O *OurGrid Peer* é o componente que gerencia, em cada domínio administrativo, as máquinas do *grid* que são disponibilizadas pelo domínio. Em geral, um nó é disponibilizado por domínio. Um nó junta-se ao *grid*, notificando um serviço de descoberta de nós sobre a sua existência e o mesmo é imediatamente informado sobre a presença de outros nós do *grid*.

O *OurGrid Broker* é responsável por fornecer aos usuários uma interface para submeterem suas aplicações. O *OurGrid Broker* também é responsável por alocar as aplicações aos nós do *grid* e por gerenciar a execução das aplicações alocadas. Um usuário que queira executar uma tarefa no *grid* deve utilizar o *OurGrid Broker* para se conectar a um nó conhecido.

Sempre que há requisições dos usuário o *OurGrid Peer* tenta satisfaze-los com os recursos disponíveis. Neste procedimento, usuários locais tem sempre maior prioridade que os usuários de outros domínios, não havendo nenhuma requisição dos usuários do domínio, os recursos disponíveis são considerados ociosos. Nesse caso, um modelo chamado *Network of Favours* é utilizado. *Network of Favours* é um modelo de alocação de recursos baseado em reputação. Os *peers* que doam mais recursos têm uma melhor reputação e, desta forma, recebem maior prioridade no momento em que requisitam recursos de outros *peers*. Com isso, evita-se o fenômeno de *free-riding*, no qual os *peers* apenas consome recursos.

2.6.2 A Framework for P2P Application Development (Walkerdine et al., 2008)

Em (Walkerdine et al., 2008), temos a definição de um framework P2P para ambiente distribuído, no qual as extensões são implementadas como *plugins* que comunicam com o framework através de uma camada específica (*Application Layer*), que são de domínios específicos, e os dados manipulados pelos *plugins* são tratados como recursos, que representam de forma geral, qualquer coisa que os *plugins* podem contribuir. O framework foi instanciado pelo autores utilizando uma aplicação de mensagem instantânea e organizado conforme mostra a figura 2.9.

Na figura 2.9, a camada P2P *Protocol/Substrate* representa o *middleware* P2P utilizado, que é abstraído pela camada *Protocol/Substrate specific Interface Layer* responsável por mapear as operações disponibilizadas pelo *middleware* para a aplicação. O *middleware* P2P empregado nesse caso é o JXTA (JXTA, 2003). A camada *Application Development Layer* fornece as operações que são comuns as aplicações P2P de comunicação como: mensagem de comunicação, operação de pesquisa, monitoramento, transferência de arquivo, lista de favoritos e uma interface gráfica para interação com o usuário.

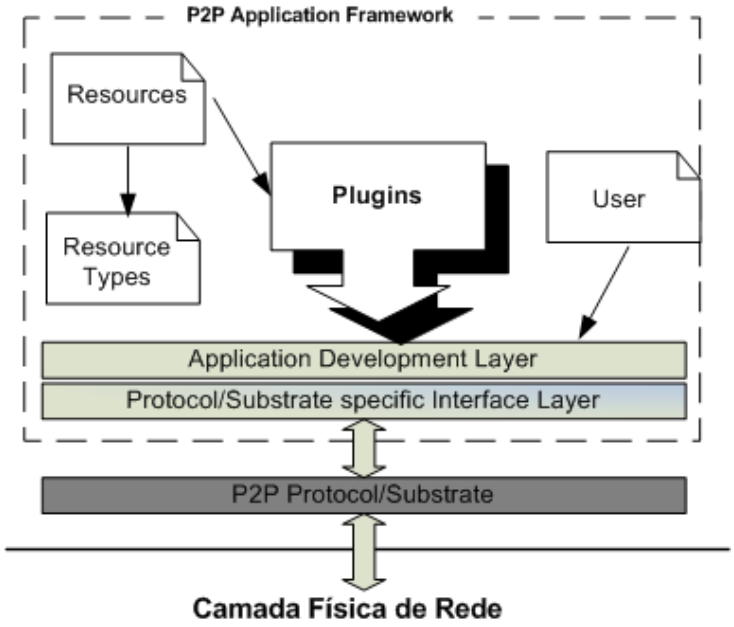


Figura 2.9: Modelo do framework P2P proposto por Walkerdine et al.(2008).

2.6.3 P2P Agent Platform for Ubiquitous Service (PIAX) (Fujiwara et al., 2008)

O P2P Interactive Agent eXtensions (PIAX) (Fujiwara et al., 2008) (figura 2.10) é uma plataforma de agentes distribuídos baseada na arquitetura P2P. O framework possui uma arquitetura hierárquica e modular com interfaces bem definidas que favorecem a modularidade e extensibilidade do framework. Nesse framework, os *overlays* P2P são implementados como *plugins* e por padrão são implementados os *overlays*: DHT (Distributed Hash Table), LL-Net (Location-based Logical P2P Network) (Kaneko et al., 2005) e *Skip Graph* (Aspnes and Shah, 2003). Além disso, o framework disponibiliza uma API para descoberta de recursos, publicada como *Web Services* SOAP e interfaces JSON-RPC.

Na figura 2.10 temos a arquitetura do framework.

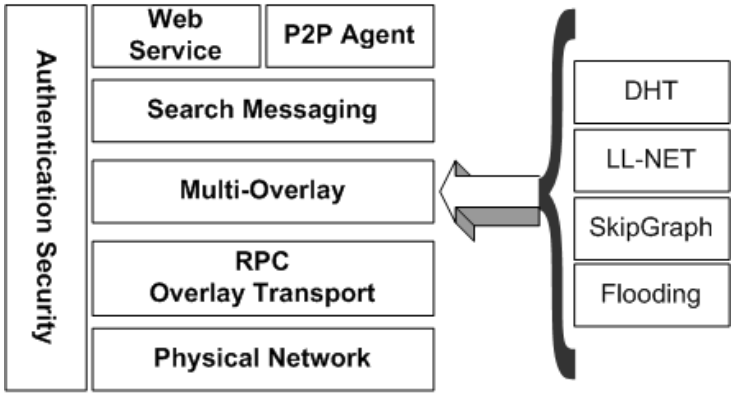


Figura 2.10: Arquitetura do PIAX Fujiwara et al. (2008)

Na figura 2.10, a camada *Physical Network* fornece os serviços de rede correspondente como comunicação TCP/UDP. A camada *RPC / Overlay Transport* oculta das camadas superiores o *middleware* de rede utilizado, tendo como foco a transparência na comunicação. A camada *Multi-Overlay* disponibiliza os *overlays* P2P a serem utilizados pelas aplicações como *plugins*. O framework disponibiliza uma API de descoberta de mensagens chamada de *discoveryCall* para ser utilizada pelas aplicações, implementada pela camada *Search Messaging*. Acima dessas camadas temos os agentes P2P (P2P Agents) e os *Web Services* SOAP e JSON. Por fim, a camada *Authentication Security*, que entrelaça todas as demais camadas, fornecendo um mecanismo para autenticação dos nós e dos agentes.

2.6.4 Quadro Comparativo

A tabela 2.3 apresenta as principais características dos frameworks P2P apresentados nesta dissertação.

Quanto ao *middleware*, (Walkerdine et al., 2008) foi implementado utilizando o JXTA (JXTA, 2003), (Cirne et al., 2006) utiliza atualmente o protocolo de troca de mensagens XMPP que também é utilizado como *overlay* e (Fujiwara et al., 2008) utiliza Web Service SOAP ou JSON. Quanto a política de alocação de tarefa, em (Cirne et al., 2006) assim como nos outros trabalhos, a política é definida pela aplicação, porém (Cirne et al., 2006) emprega o modelo *Network of Favours* para executar as aplicações. Apenas (Fujiwara et al., 2008) implementa múltiplos *overlays* e, quanto ao modelo P2P, apenas (Fujiwara et al., 2008) é totalmente descentralizado, sendo que, (Walkerdine et al., 2008) é centralizado e (Cirne et al., 2006) é hierárquico.

Na última linha do quadro ilustrado na tabela 2.3 são apresentadas as características do framework proposto na presente dissertação, que serão discutidas em detalhe no capítulo 4.

Paper	*Middleware* P2P	Política de Alocação de Tarefas	*Overlay*	Organização
(Cirne et al., 2006)	XMPP	Definida pelo usuário	XMPP	Hierárquico
(Walkerdine et al., 2008)	JXTA	Definida pelo usuário	Aplicação	Centralizado
(Fujiwara et al., 2008)	RPC	Definida pelo usuário	DHT, LL-NET, Skip Graph, Flooding	Descentralizado
Framework proposto	Múltiplos	WSR, LWSR e Definida pelo usuário	Chord / Definido pelo usuário	Descentralizado

Tabela 2.3: Quadro comparativo dos frameworks

Capítulo 3

Políticas de Alocação de Tarefas em Ambientes Distribuídos

Para que uma aplicação seja executada em um ambiente distribuído, ela é geralmente "quebrada" em unidades menores chamadas tarefas. Potencialmente, cada tarefa pode ser alocada para execução em um nó distinto, respeitados os requisitos mínimos de CPU, memória, sistema operacional, entre outros. Dessa maneira, tira-se proveito do paralelismo existente entre tais ambientes, para que a aplicação como um todo seja executada de maneira mais rápida.

A alocação de tarefas a nós determina, portanto, em que nó cada tarefa será executada. Em um problema de alocação de tarefas geralmente não existem restrições de precedência temporal entre as mesmas (Casavant and Kuhl, 1988). Por outro lado, as políticas de escalonamento determinam o mapeamento nó versus tarefa, considerando as restrições temporais em um Grafo Acíclico Dirigido (DAG).

Uma tarefa pode ser definida como uma entidade que pode ser executada por qualquer nó utilizando de mensagens para obter dados e para comunicar com outras tarefas através da rede (Chang and Oldham, 1995). O objetivo da alocação de tarefa é maximizar a utilização dos recursos computacionais assegurando alto nível de desempenho aos processos (Chang and Oldham, 1995) e constitui um problema NP-Completo (Graham et al., 1979). Por essa razão, diversas heurísticas foram propostas para solucioná-lo.

3.1 Taxonomia de Casavant

Para Casavant and Kuhl (1988) os termos escalonamento e alocação são implicitamente distintos na literatura. Entretanto, pode-se argumentar que esses termos são meramente formulações alternativas para o mesmo problema, com alocação vista em termos da alocação de recursos (do ponto de vista do recurso) e escalonamento visualizado do ponto de vista do consumidor. Nesse caso, alocação e agendamento são dois termos que descrevem o mesmo mecanismo, mas, de ponto de vistas diferentes (Casavant and Kuhl, 1988).

Nesse caso, a função de alocação de tarefas pode ser vista como inserida dentro do mecanismo de gerenciamento de recursos. O gerenciamento de recursos é basicamente um mecanismo ou política utilizada para gerenciar de forma eficaz e

eficiente o acesso e o uso dos recursos pelos vários consumidores, conforme apresentado na figura 3.1.

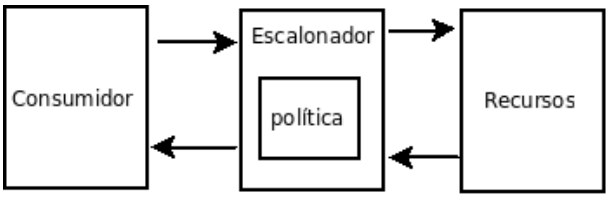


Figura 3.1: Sistema de alocação. (Casavant and Kuhl, 1988)

Em Casavant and Kuhl (1988) os autores apresentam uma taxonomia das características do escalonamento de tarefas representadas na figura 3.2.

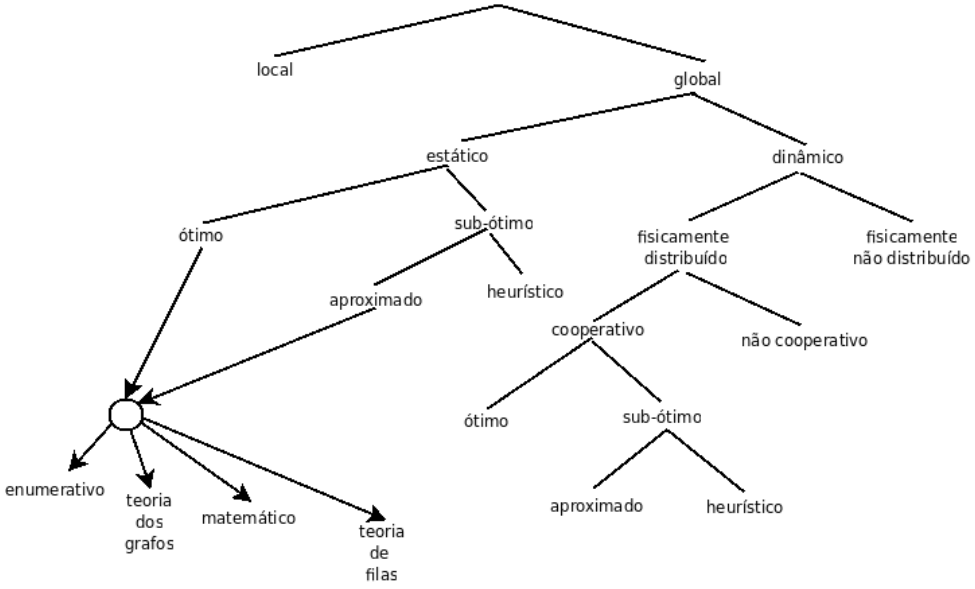


Figura 3.2: Características do escalonamento de tarefas. (Casavant and Kuhl, 1988)

Nessa classificação o escalonamento pode ser estático ou dinâmico, dependendo do momento em que as decisões para a alocação de tarefas são tomadas. No escalonamento dinâmico as decisões são tomadas durante a execução do programa e no escalonamento estático as decisões são tomadas antes da execução do programa.

O escalonamento estático ou também conhecido como escalonamento determinístico (Lo, 1988) pode ser classificado como ótimo ou sub-ótimo. Nos casos em que todas as informações sobre o estado dos sistemas são conhecidas, assim como, os recursos necessários, uma atribuição ótima pode ser realizada baseada em algum critério. Nos casos em que a complexidade do problema não é totalmente conhecida, pode-se empregar soluções sub-ótimas para obter respostas em tempos inferiores ao exponencial. Os escalonadores sub-ótimos ainda podem ser classificados em aproximados ou heurísticos. O escalonamento aproximado ocorre quando o algoritmo executado é o algoritmo ótimo, porém sua execução é terminada quando uma

solução definida como "boa" é atingida. O escalonamento heurístico utiliza critérios identificados empiricamente ou intuitivamente como responsáveis por algum tipo de desempenho no escalonamento (Park et al., 1997).

Uma vantagem do escalonamento estático é a ausência de sobrecarga para determinar o escalonamento em tempo de execução. No entanto, a principal desvantagem é a falta de adequabilidade às plataformas usuárias, devido a exigência de conhecer previamente o comportamento da aplicação em relação a plataforma utilizada (Park et al., 1997).

Para Casavant and Kuhl (1988) a política de alocação pode ser classificada como centralizada ou distribuída baseada em dois componentes, responsabilidade e autoridade. Quando a responsabilidade da escolha da política de alocação é compartilhada entre as entidades do sistema distribuído a alocação é dita distribuída. Quando a autoridade é distribuída entre as entidades de gerenciamento do sistema, ela é dita descentralizada (Casavant and Kuhl, 1988).

3.2 Classes de Tarefas

Seja P um conjunto de nós, $P = \{P_1, \cdots, P_n\}$ onde n é o número de nós disponíveis para a execução de tarefas e T um conjunto de tarefas, $T = \{T_1, T_2, \cdots, T_m\}$, onde m é o número de tarefas a serem executadas, x_{iq} o custo de execução de uma tarefa t_i quando é atribuída e executada em um nó P_q, $1 \leq i \leq k$, $1 \leq q \leq n$; e C_{ij} o custo de comunicação entre duas tarefas $T_i \; e \; T_j$ quando atribuídas a processadores diferentes (Graham et al., 1979; Lo, 1988). Formalmente, a atribuição de tarefas à nós pode ser descrita por uma função de um conjunto de tarefas a um conjunto de nós, $f : T \leftarrow P$. Em um sistema com n nós e m tarefas, a alocação de tarefas consiste em atribuir as tarefas aos nós de modo que uma solução deve ser selecionada em n^m possibilidades de atribuições (Graham et al., 1979; Lo, 1988).

Quando uma aplicação é composta de tarefas independentes é dita aplicação *Bag-of-Task(BoT)* (Cirne et al., 2003) que possui a característica de serem facilmente adaptável para rodar em uma plataforma distribuída (Assis et al., 2006).

Em um ambiente distribuído a alocação de tarefas aos nós é realizada por um software denominado escalonador responsável por selecionar os recursos mais apropriados para submeter as tarefas da aplicação para execução. Um escalonador utiliza de heurísticas para realizar as decisões de escalonamento.

Na literatura há diferentes políticas de alocação de tarefas. Dentre as políticas centralizadas temos: Fixed (Static Scheduling) (Shao, 2001), Self-Scheduling(SS) (Tang and Yew, 1986), Trapeizodal Self-Scheduling (TSS) (Tzen and Ni, 1993), Guided Self-Scheduling (GSS) (Polychronopoulos and Kuck, 1987) Factoring (FAC2) (Hummel et al., 1992), Weighted Factoring (Hummel et al., 1996) dentre outras, que possuem a características de implementarem uma política centralizada utilizando-se de um servidor central para manter uma visão global do sistema. Quanto as políticas de alocação descentralizadas temos as apresentadas em (Mendes et al., 2009), CSAM (Chang and Oldham, 1995), (Yau and Satish, 1993), (Shen and Yuan, 2008).

3.2.1 Aplicações *Parameter Sweep*

Uma aplicação *parameter sweep* pode ser definida como um conjunto $T\{t_1, t_2, \cdots, t_n\}$ de n tarefas seqüencialmente independentes. Nesse contexto, independência está atrelada ao fato de que não há nenhum tipo de comunicação ou relação de precedência entre as tarefas que compõem uma aplicação. Além disso, todas as n tarefas realizam o mesmo tipo de processamento. Nesse caso, a diferença entre duas tarefas quaisquer, t_1 e t_2, são os parâmetros de entrada de cada tarefa (Casanova et al., 2000).

As aplicações *parameter sweep* são, em geral, aplicações desenvolvidas para explorar um grande espaço de possibilidades de resolução de um problema. Cria-se várias tarefas, cada uma resolvendo o mesmo problema, mas com o conjunto de parâmetros de entrada distintos, que juntas abrangem o espaço de parâmetros possíveis. Por serem tarefas independentes, esse tipo de aplicações são ideais para serem executadas em plataforma de *grid*, onde a distribuição geográfica dos nós pode implicar em altos custos de comunicação. Em Casanova et al. (2000), temos um exemplo de escalonador de tarefas projetado para aplicações do tipo *parameter sweep*.

3.2.2 Aplicações *Bag-Of-Tasks*

Aplicações *Bag-Of-Task* (BoT) podem ser entendidas como sendo um tipo de generalização das aplicações do tipo *parameter sweep*. As aplicações BoT são aplicações paralelas compostas por n tarefas independentes uma das outras, no sentido de que não há uma relação de precedência entre as tarefas e também não há nenhuma comunicação entre elas (Cirne et al., 2003). Porém, ao contrário das aplicações *parameter sweep*, nas aplicações *Bag-Of-Tasks* não há garantias de que as tarefas que compõem a aplicação executem o mesmo tipo de processamento. Nesse caso, duas tarefas quaisquer, t_1 e t_2, podem realizar processamentos completamente distintos uma da outra.

Esse tipo de aplicação pode ser facilmente adaptada para executar em plataforma de *grid*, bastando apenas alocar as tarefas da aplicação à diferentes nós do *grid*. Isso ocorre devido ao fato dessas aplicações não necessitarem de comunicação entre elas, e dessa forma, não sofrem grandes impactos por conta de uma possível latência de comunicação entre os nós do *grid*. Nesse caso, assim como as aplicações *parameter sweep*, esse tipo de aplicações são ideais para executarem em plataforma de *grid*.

Aplicações BoT são utilizadas em uma variedade de aplicações como: *parameter sweep* (Abramson et al., 2000), computação biológica (Balls et al., 2004), renderização de imagens (Smallen et al., 2000), dentre outras. Em Cirne et al. (2003), temos um exemplo de escalonador projetado para esse tipo de aplicações.

3.2.3 Aplicações de Workflow

Uma aplicação de *workflow* é uma aplicação composta por um conjunto de tarefas que devem ser executadas segunda uma ordem parcial determinada por dependência de controle e de dados (Cooper et al., 2004). Essa classe de aplicação representa

todas as aplicações que podem ser descritas por meio de um Grafo Acíclico Dirigido (DAG).

Em Cooper et al. (2004), temos um exemplo de escalonador projetado para aplicações do tipo *workflow*.

3.3 Alocação de Tarefas Centralizadas

Em um modelo de aplicação mestre/escravo composta por um mestre m e S escravos, $\{S_1, \cdots, S_i\}$, o mestre é responsável por controlar a distribuição das N tarefas e coleta dos resultados aos i escravos, que executam as tarefas designadas pelo mestre. Nesse modelo, o número de escravos controlados por um mestre depende da carga de trabalho do mestre e da sua disponibilidade de recursos, e o número de tarefas que são atribuídas a um escravo S_i é determinado pela função $A(N, S)$, onde $A(N, S)$ representa uma política de alocação específica.

A seguir, apresentamos políticas de alocação de tarefas muito usadas em ambientes distribuídos.

3.3.1 Fixed (Static Scheduling)

A política de alocação *Fixed* (*Static Scheduling*) (Shao, 2001) é adequada para sistemas homogêneos com recursos dedicados e aplicações que não apresentam variações de comportamento, em tempo de execução. Nesse caso, uma abordagem estática é empregada, dividindo uniformemente todas as N tarefas aos nós escravos. Com isso, a equação da política de alocação *Fixed* pode ser definida como:

$$A(N, P) = \frac{N}{P} \tag{3.1}$$

Uma desvantagem dessa estratégia de alocação é que nem sempre é possível quantificar precisamente a quantidade de trabalho necessário para um problema. Nesse caso, erros na previsão dos parâmetros podem resultar em erros de superestimar a carga de trabalho de alguns nós e de subestimar outros, resultando em um aumento de tempo de execução, pois a tarefa cujo tempo de execução foi subestimado irá demorar mais para completar, enquanto o processo superestimado fica ocioso.

3.3.2 Self-Scheduling (SS)

A estratégia de alocação *Self-Scheduling(SS)* (Tang and Yew, 1986) distribui as tarefas à medida que elas vão sendo solicitadas pelos nós escravos. A equação 3.2 apresenta a função de alocação da estratégia *Self-Scheduling(SS)*. Quando o nó termina o processamento dessa tarefa, uma nova tarefa é solicitada ao nó mestre, até que não haja mais tarefas.

$$A(N, P) = 1 \; enquanto \; existir \; tarefas \; a \; serem \; alocadas. \tag{3.2}$$

Nessa estratégia, a cada nó é alocado exatamente uma tarefa. No SS, o máximo de tempo que um conjunto de nós podem ficar ociosos é determinado pelo tempo de processamento de uma tarefa pelo escravo mais lento. Porém, uma desvantagem da estratégia *Self-Scheduling* é o grande número de comunicações entre o nó escravo e o nó mestre.

3.3.3 Guided Self-Scheduling (GSS)

A estratégia *Guided Self-Scheduling* (GSS) (Polychronopoulos and Kuck, 1987), aloca tarefas em grupos que diminuem de tamanho exponencialmente ao longo do tempo. Nessa estratégia, o tamanho dos blocos em uma iteração é $\frac{1}{P}$ do total de tarefas restantes. A equação 3.3 apresenta a função de alocação da estratégia GSS, onde s representa o estágio da alocação de tarefas.

$$A(s, N, P) = max(\lfloor \frac{N(1-\frac{1}{p})^{s-1}}{P} \rfloor, 1), s > 0 \tag{3.3}$$

A *Guided Self-Scheduling* é uma estratégia onde o número de tarefas alocadas aos nós diminui ao longo do tempo. Nessa estratégia, o tamanho das alocações inicia-se relativamente grande e vai diminuindo exponencialmente a cada nova alocação. A desvantagem é que, em um ambiente heterogêneo, pode ocorrer de serem atribuídos blocos grandes de processamento a nós com pouco poder computacional, causando um desbalanceamento de carga no tempo final do processamento.

3.3.4 Package Weighted Adaptive Self-Scheduling (PSS)

O *Package Weighted Adaptive Self-Scheduling* (PSS) (Sousa and Melo, 2006) é uma estratégia de alocação de tarefas que adapta a escolha da política de alocação de tarefas para ambiente em *grid*, capaz de modificar o tamanho dos blocos de alocação em tempo de execução levando em consideração a heterogeneidade do *grid* e as características dinâmicas de seus nós. Essa estratégia define o esquema de atribuição de pesos aos nós do *grid* de maneira genérica, viabilizando a utilização de qualquer função de alocação.

A equação 3.4 apresenta a função de alocação dessa estratégia. Nessa equação, $A(N, P)$ é a política de alocação para um sistema com N tarefas e P nós e $\Phi(m, p_i, P)$ é a média ponderada do desempenho das últimas Ω notificações de execução de cada nó p_i do *grid* (Sousa and Melo, 2006).

$$A(m, p_i, N, P) = A(N, P) * \Phi(m, p_i, P) \tag{3.4}$$

O PSS foi implementado em um framework de alocação adaptativa de tarefas para execução do BLAST (*Basic Local Alignment Tool*) (Altschul et al., 1990) em ambiente em *grid* - o packageBLAST (Sousa and Melo, 2006), que realiza a comparação de seqüências biológicas replicando o banco de dados em cada nó do *grid* em um modelo mestre/escravo.

Atualmente, a implementação conta com cinco estratégias de alocação de tarefas com a possibilidade do usuário incluir a sua própria estratégia de alocação. Uma das desvantagens desse modelo é que ele ainda é centralizado e não é possível o usuário modificar a estratégia de alocação em tempo de execução.

3.3.5 Work Stealing

Work Stealing (Blumofe and Leiserson, 1999) é uma estratégia muito popular de alocação de tarefas, proposta inicialmente para multiprocessadores com memória compartilhada, no contexto de aplicações *multi-threads* que podem ser expressas como um Grafo Acíclico Dirigido (DAG). Em *Work Stealing*, cada processador tem uma fila duplamente encadeada (*dequeue*) e, quando um processador fica ocioso, ele "rouba" uma tarefa de algum processador escolhido aleatoriamente, removendo a tarefa roubada do final da fila do processador escolhido, desde que a fila não esteja vazia.

Para a política de *Work Stealing*, alguns resultados teóricos importantes foram obtidos, tais como, o tempo de execução e o número de tentativas de roubos (Blumofe and Leiserson, 1999). Além disso, a estratégia de *Work Stealing* se mostrou estável, no sentido de que a comunicação só é iniciada quando os nós estão ociosos e, portanto, o desequilíbrio ocorre se o nó roubar cerca de metade do trabalho em qualquer tempo.

Em (Michael et al., 2009), temos uma versão modificada da estratégia de *Work Stealing* para sistemas multiprocessados, onde cada tarefa é executada pelo menos uma vez.

Para ambientes distribuídos, temos algumas tentativas de uso da estratégia de *Work Stealing*. Em (Kumar et al., 1994), temos uma proposta de framework para avaliar o balanceamento de carga de alguns algoritmos em arquitetura paralelas com memória distribuída e com processadores iguais. Os resultados mostram que a estratégia de *Work Stealing* obtêm muito bons *speedups* para arquiteturas com 1024 processadores. Em (van Nieuwpoort et al., 2001), a política *Cluster Aware Random Stealing* (CSR) é apresentada. Nesse caso, um roubo é realizado aleatoriamente entre os nós que compõem o *cluster* e os roubos entre diferentes *clusters* são realizados de forma assícrona e descentralizado. Em (Blumofe and Lisiecki, 1997), os autores propõem a execução de tarefas *Cilk* com *Work Stealing* executando em rede de estações de trabalho. Nesse caso, um *Cilk* é um conjunto de várias outras tarefas *Cilk*, cada uma executando em um nó diferente. Cada *Cilk* tem um mestre, chamado *cleavinghouse*, responsável por manter as informações sobre os escravos. O conjunto de escravos é dinâmico e aumenta a medida que os nós vão ficando ociosos e, diminui, a medida que alguma atividade local é detectada. A tolerância a falhas é realizada através de *checkpoint* e migração das tarefas.

3.3.6 Random Walking

A estratégia de *Random Walking* (Gkantsidis et al., 2006) é um algoritmo de alocação de tarefas em sistemas P2P não estruturados. Nesse caso, um nó envia um pedido de consulta para execução de tarefa a um nó vizinho escolhido aleatoria-

mente. Se o nó selecionado está ocioso, ele aceita a tarefa. Caso contrário, o pedido é encaminhado a outros nós vizinhos, selecionados aleatoriamente. O pedido de execução de tarefa geralmente tem como parâmetros um valor de TTL, que restringe o número de vezes que o pedido é encaminhado.

3.4 Alocação de Tarefas Distribuídas

Ao contrário das políticas centralizadas discutidas na seção 3.3, as políticas distribuídas não requerem a existência de um nó mestre, que controla o estado global do sistema.

Em um ambiente distribuído e descentralizado, a alocação de tarefas constitui-se um problema complexo, uma vez que não há uma visão global do sistema, que inclui a informação de quantos e quais são os nós de computação que formam a rede. Além disso, diferentes classes de aplicações distribuídas podem exigir a utilização de diferentes políticas de alocação de tarefas, que podem de alguma maneira, depender de conhecimento sobre a rede.

Nesse caso, utilizam-se de heurísticas para controlar o estado do sistema, sem a presença de um nó centralizador.

A seguir, apresentamos algumas políticas de alocação de tarefas distribuídas.

3.4.1 Adaptive Resource Management (Repantis et al., 2005)

Em (Repantis et al., 2005), os autores apresentam uma arquitetura descentralizada para gerenciamento de recursos em sistemas distribuídos de larga escala. Nessa arquitetura, um sistema distribuído é modelado como uma rede de nós de computação agrupados em domínios de acordo com a sua proximidade topológica utilizando gerenciadores de recursos de domínio, selecionados entre os nós, para agirem como líderes do domínio.

Os nós são divididos em domínios geográficos. O parâmetro que determina o tamanho do domínio é o número máximo de nós que um gerenciador de recurso pode gerenciar, que depende da sua capacidade. Cada gerenciador de domínio é conectado a todos os nós de seu domínio, de modo que o gerenciador de recurso possua uma visão global do domínio em termos de aplicação e utilização dos recursos. Cabe ao gerenciador de domínio distribuir as tarefas aos nós que atendem os requisitos de qualidade (*QoS*) e por coletar as informações dos nós do domínio utilizados para alocar as tarefas e sobre os nós que estão disponíveis.

Associado ao gerenciador de recursos tem-se o gerenciador de conexão, responsável por gerenciar as conexões dos nós e o *Profile*, que é responsável por mensurar a carga de trabalho da rede, dos nós, além de monitorar a quantidade de trabalho executada por cada processo, assim como, a sua comunicação. Nesse modelo, as informações são mantidas em três níveis. No nível de domínio, os gerenciadores de recursos mantêm informações sobre todos os nós do domínio e sobre as aplicações finais dos usuários, coletadas em tempo de execução. No nível de sistema, os gerenciadores de domínio mantêm informações sumarizadas atualizadas sobre os outros

domínios, utilizando o protocolo *gossiping*. No nível dos nós, cada nó mantém informações sobre os recursos alocados e sobre as aplicações executadas localmente.

As tarefas das aplicações são modeladas como uma seqüência de invocações de objetos e serviços. As tarefas podem ser executadas concorrentemente por diferentes usuários no sistema, representadas como um grafo G_s, apresentado na figura 3.3. Nesse grafo, os vértices representam os objetos ou os serviços do sistema e as arestas, as conexões que foram estabelecidas entre os nós para receberem os serviços.

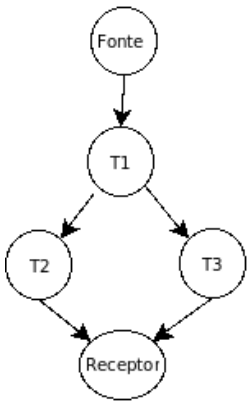


Figura 3.3: Exemplo de grafo de serviços (Repantis et al., 2005).

O gerenciador de recurso mantém as seguintes informações sobre cada tarefa t de uma aplicação:

- *Tempo*: o tempo necessário, especificado pelo usuário, para a execução completa da tarefa;
- *Importância*: uma métrica, especificada pelo usuário, que representa uma importância relativa da aplicação;
- *Tempo de execução*: o tempo total estimado para finalizar uma tarefa t, computado como a soma do tempo de processamento dos objetos e dos serviços em cada processo, mais o tempo depreendido na comunicação até o início da execução da tarefa.

O gerenciador de recurso também mantém um grafo de recursos G_r para seus domínios D, representado na figura 3.4. O grafo G_r é um grafo direcionado, onde cada vértice v representa um estado da aplicação, e cada aresta e um serviço, acompanhado da sua respectiva carga de trabalho. O objetivo do grafo de recursos é identificar os serviços disponíveis em cada nó, de modo a alocá-los às aplicações requisitantes.

Cada gerenciador de recurso é responsável por construir o grafo dos serviços da aplicação, G_s de modo que os requisitos de qualidade sejam atendidos e que a carga de trabalho seja distribuída igualmente entre os nós de seu domínio. Para avaliar como a carga de trabalho pode ser distribuída entre os nós de um domínio, o gerenciador de recurso utiliza da métrica *Fairness* (Jain et al., 1984), definida na equação 3.5.

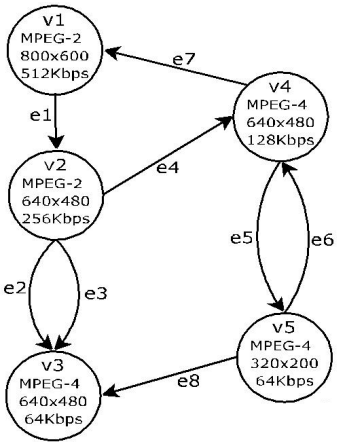


Figura 3.4: Exemplo de grafo de recursos (Repantis et al., 2005).

$$F(\bar{l}_{p_D}) = \frac{(\sum_{p \in P_D} l_p)^2}{|P_D| . \sum_{p \in P_D} l_p^2} \tag{3.5}$$

Na equação 3.5, P_D é o conjunto de nós em um domínio D e l_p a carga de trabalho de um nó p. Essencialmente, o índice *Fairness* é um índice absoluto que quantifica a distribuição uniforme $(\bar{l}_{p_D})$ entre os nós, P_D, de um domínio (Repantis et al., 2005).

A alocação de tarefas é realizada da seguinte maneira. Primeiramente o usuário submete uma tarefa ao sistema, informando os requisitos de qualidade. A partir disso, o gerenciador de recursos tenta alocar os recursos que atendem os requisitos de *QoS* especificados e que maximizem a distribuição de carga. Para isso, o gerenciador de recursos utiliza do grafo de recursos para procurar por configurações no domínio que satisfazem os parâmetros requisitados. Caso nenhum nó seja encontrado, o algoritmo retorna informando que não foi encontrado nenhum nó que atenda os requisitos especificados.

3.4.2 Organic Grid (Chakravarti et al., 2005)

Em (Chakravarti et al., 2005), os autores apresentam uma arquitetura para escalonamento e execução de tarefas em ambiente em *grid*, onde os nós são organizados em uma estrutura de árvore e cada nó possui autonomia para adicionar ou remover nós filhos de acordo com alguns critérios, como desempenho ou falha de comunicação.

Segundo (Chakravarti et al., 2005), estruturar a topologia da rede como uma árvore possibilita uma certa autonomia e um comportamento descentralizado aos nós, sem requerer conhecimento aprofundado sobre a configuração dos nós participantes, da topologia da rede ou sobre a largura de banda da conexão. Os nós são encapsulados como agentes com autonomia para distribuir os dados pela rede, para mover-se dinamicamente pelos *hosts* de acordo com a disponibilidade dos recursos,

ao mesmo tempo que mantêm comunicação com os outros agentes. Nessa configuração, os nós são organizados de modo a maximizar a comunicação e a distribuição das tarefas entre os nós. A efetividade do desempenho do sistema é mensurada e ajudada através do *feedback* dado pelos nós filhos aos seus nó pai.

A arquitetura possui foco no escalonamento de aplicações *Bag-of-Tasks* (seção 3.2.2), com os dados localizados localmente. Com isso, uma aplicação t atribuída a um agente pode ser dividida em n tarefas independentes, que são executadas seqüencialmente pelo agente. Cada agente também é preparado para receber requisições por tarefas de outros nós. Caso um nó tenha tarefas a serem executadas, e receba uma requisição de outro nó, o nó manda uma cópia de si próprio para o nó requisitante que torna-se um nó filho na estrutura da árvore.

O nó filho solicita continuamente ao seu nó pai mais tarefas, quando todas as suas tarefas já tiverem sido executadas. Mesmo quando o número de tarefas disponíveis para serem enviadas pelo nó pai é menor que o solicitado pelo nó filho, o nó pai responde com o número disponível de tarefas restantes para execução. Com isso, o nó pai mantém o número de subtarefas pendentes a serem enviadas, e envia uma requisição para os seus nós filhos.

Dessa forma, a topologia da rede resulta em uma árvore onde a raiz é o nó de origem da requisição, e os filhos são criados à medida que forem entrando em contato com o nó raiz. Isso proporciona o escalonamento e execução de tarefa de forma descentralizada, e uma certa autonomia aos nós participantes da rede.

3.4.3 Alocação de Tarefas em Sistemas P2P Não-Estruturados (Awan et al., 2006)

Em (Awan et al., 2006), os autores apresentam uma arquitetura de sistema P2P não-estruturado para compartilhamento distribuído de ciclos entre *host* de internet. A arquitetura fundamenta-se em um algoritmo de caminhamento randômico para distribuição de carga, com suporte a tolerância a falhas e serviços probabilísticos para monitorar a execução das tarefas e a contribuição de cada nó participante, de forma descentralizada.

A tolerância a falhas é obtida através da redundância das tarefas, de modo que quando uma aplicação t necessita ser executada, ela é quebrada em k tarefas independentes e replicadas através de um fator $r \geq 1$. Nesse caso, um valor de $r = 2$ implica que um lote de tarefas deve ser atribuído a dois nós para ser executado. Com isso, o algoritmo de submissão dos lotes de tarefas apresenta o seguinte funcionamento: (Awan et al., 2006)

1. Qualquer *host* que necessita submeter a sua aplicação de tamanho k, deve-se submeter as tarefas e o valor do fator de replicação, r;
2. Para cada tarefa, o *host* deve selecionar um nó de modo aleatório e submeter a tarefa, juntamente com o fator r;
3. Cada nó, ao receber a tarefa, decrementa o valor de r em uma unidade e, se $r > 0$, envia uma cópia da tarefa para outro nó, também escolhido de modo aleatório.

Para procurar por um recurso na rede percorrendo um número aleatório de nós, sem que seja preciso manter em cada nó informações sobre todos os outros nós, a arquitetura utiliza do modelo de cadeia de Markov, definido sobre um espaço de estados e sobre uma matriz de transição de estado (Awan et al., 2006). A rede de nós forma o espaço de estados e a probabilidade de mover de um nó para qualquer um dos seus vizinhos, determina a transição.

Com isso, um sistema distribuído é representado como um grafo não direcionado $G(V, E)$ com $|V| = N$ nós e $|E| = e$ conexões, com o número de conexões de um nó denotado por d_i, para $1 \leq i \leq N$, o número de vizinhos de um nó determinado por $\tau(i)$, onde a aresta $(i, j) \in E, \forall j \in \tau(i)$ e $P = \{p_{ij}\}$ a probabilidade de mover de um nó i para o nó j, em um passo.

A probabilidade das transições de cada nó i da rede é obtida através de um algoritmo que objetiva distribuir uniformemente a probabilidade de se caminhar na rede.

3.4.4 Distribuição de Tarefas em uma Rede *n-Cycle* (Bölöni et al., 2006)

(Bölöni et al., 2006) propõem a arquitetura *n-Cycle* para alocação de tarefas em ambiente distribuído descentralizado, empregando práticas de economia de mercado, onde os nós, provedores p, ofertam serviços (execução das tarefas), aos cliente c, com um custo $custo(p, c)$, igual para todos os nós.

Seja um conjunto de tarefas $T = \{t_1, t_2, \cdots, t_k\}$. Um conjunto de provedores $P = \{p_1, p_2, \cdots, p_i\}$ é provedor das tarefas t_k se qualquer provedor p_i pode executar qualquer tarefa t_k e se a diferença de desempenho para executar qualquer tarefa t_k não for considerável. Isso significa que qualquer cliente c que deseja executar uma tarefa t_k pode escolher qualquer provedor, pois o custo de execução será o mesmo, ou seja, o preço entre os provedores é uniforme, assim como o tempo de execução entre os provedores é constante.

Na arquitetura *n-Cycle* as requisições para execução das tarefas são encaminhadas ao longo da rede até que um provedor disponível seja encontrado. Para (Bölöni et al., 2006), o número de saltos necessários para que uma tarefa seja alocada é o principal critério de desempenho do sistema. Os dois principais componentes dessa arquitetura são: (a) criação e manutenção do modelo *n-Cycle* e (b) a política de encaminhamento juntamente com a manutenção dos dados ao longo dos encaminhamentos.

A rede *n-Cycle* é formada por *links* direcionados onde, para qualquer *link* $A \rightarrow B$, a tarefa é encaminhada de A para B, enquanto o sentido da informação sobre a execução da tarefa é de B para A. Os *links* formam n ciclos conectando todos os nós do sistema. Para qualquer ciclo da rede, todo nó está conectado à n nós no sentido do *link* e n nós no sentido contrário ao *link*. Dessa forma, as tarefas são encaminhadas pelos nós no sentido dos *links* e a informação sobre a execução é enviada no sentido contrário. A figura 3.5 apresenta uma rede com 5 nós e 3 ciclos.

Na figura 3.5, o primeiro ciclo é formado pelos nós $ABCDE$, apresentado pelas linhas contínuas, o segundo pelos nós $ACEBD$, linhas pontilhadas e o terceiro pelos nós $ADCEB$, linhas tracejadas.

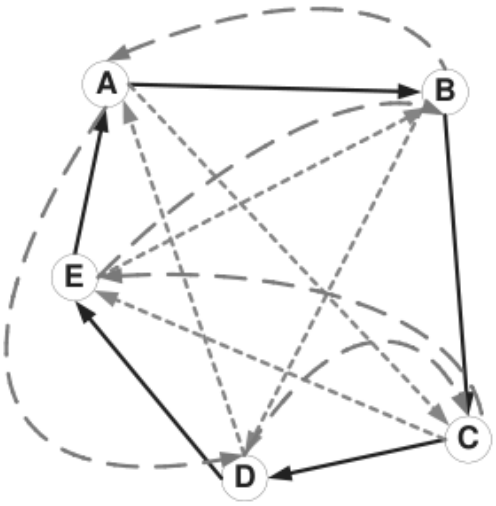


Figura 3.5: Exemplo de uma rede *n-Cycle* com 5 nós (Bölöni et al., 2006).

Nessa arquitetura, as tarefas são distribuídas através de duas classes de algoritmos: os algoritmos *stateless* que não mantêm qualquer informação sobre os nós, baseando unicamente nas regras de encaminhamento e na estrutura da rede para distribuir as tarefas, e os algoritmos *stateful* que mantêm informações sobre o estado da rede e as utilizam, para encaminhar as tarefas. Essas duas classes de algoritmos são implementadas na arquitetura através dos algoritmos: *Random Wandering* e *Weighted Stochastic Forwarding* respectivamente.

No algoritmo *Random Wandering*, a tarefa a ser alocada é atribuída a um nó apenas se o nó estiver livre. Caso contrário, o nó encaminha a tarefa para qualquer um dos nós dispostos no sentido do nó na rede de ciclos. Nesse caso, o número de saltos necessários para que uma tarefa seja aceita depende da média da carga de trabalho da rede l, sobre o número de nós N. Segundo (Bölöni et al., 2006) essa média é menor que $(1 - l)^h$ saltos.

O algoritmo *Weighted Stochastic Forwarding* (WSF) utiliza de informações coletadas pelos nós que estão abaixo e no mesmo sentido para encaminhar as tarefas. Nesse caso, todo nó mantém o seu peso w que representa a oportunidade do nó como candidato a receber uma tarefa encaminhada por outro nó. Esse peso w é composto pela habilidade do nó em receber uma tarefa para execução e pelos pesos dos nós que estão abaixo e na mesma direção do nó (Bölöni et al., 2006).

Uma tarefa é aceita pelo nó apenas se o nó estiver livre. Caso contrário, a tarefa é encaminhada aos nós inferiores ao nó, com probabilidade proporcional ao peso w de cada nó. O número de saltos é limitado pelo valor de *time-to-live* (TTL) que é decrementado a cada salto. Se o valor de TTL for igual a zero e a tarefa não estiver sido aceita por nenhum nó, a tarefa é devolvida ao cliente como rejeitada.

3.4.5 Alocação Adaptativa de Tarefas em Serviços P2P (Shen and Yuan, 2008)

(Shen and Yuan, 2008) propõem uma estratégia para alocação de tarefas de Serviços Web P2P de forma descentralizada, baseada em requisitos não funcionais, utilizando de ontologias para descrever as características dos serviços disponíveis em cada nó.

Para avaliar as propriedades não funcionais dos serviços são utilizados três conceitos: *PreferedValueType*, *Weight* e *Unified Value*. *PreferedValueType* pode assumir os valores alto ou baixo que são utilizados para identificar quantitativamente dois tipos de propriedades diferentes, por exemplo, reputação, tempo de resposta e disponibilidade. O *Weight* indica a importância e a prioridade de certas propriedades durante a execução dos serviços. O *Unified Value* indica numericamente a qualidade de cada nó calculado de acordo com a equação 3.6, de modo a ordenar a capacidade dos nós em atenderem as necessidades do requisitante do serviço.

$$UV(i) = \sum_{j=1}^{n}(PR(i,j)) * W(j)), i,\ i = 1 \cdots m \tag{3.6}$$

$$PR(i,j) = \frac{nf(i,j) - nf(min)}{nf(max) - nf(min)} \tag{3.7}$$

$$PR(i,j) = \frac{nf(max) - nf(i,j)}{nf(max) - nf(min)} \tag{3.8}$$

Nas equações 3.6 e 3.7 $PR(i,j)$ representa um valor da propriedade j do nó i, onde nf é um acrônimo para o termo não funcional. Sendo assim, $nf(max)$ e $nf(min)$ são os valores máximos e mínimos, respectivamente, da propriedade j e w o peso associado a cada propriedade. No caso em que o valor de *PreferedValueType* é "alto", o valor de $PR(i,j)$ é determinado pela equação 3.7, caso contrário, $PR(i,j)$ é determinado pela equação 3.8.

Nessa estratégia, a seleção e composição dos nós ocorre de modo a escolher o melhor caminho através da composição dos serviços.

3.4.6 Alocação de Tarefas P2P em Estruturas *Range-Queriable* (Mendes et al., 2009)

Em (Mendes et al., 2009), os autores apresentam uma arquitetura P2P capaz de executar aplicações *BoT* (seção 3.2.2) com controle totalmente distribuído, utilizando *Skip graphs* (Aspnes and Shah, 2003) para representar os nós (*machine providers*) e tarefas (*tasks providers*). Uma *machine provider* representa uma máquina com poder computacional disponível enquanto uma *task provider* é uma tarefa a ser executada e são representadas no grafo com os nomes *"mach-x"* e *"task-y"* respecti-

vamente, com x e y obtidos a partir do cálculo de *hash* do nome da máquina ou do conteúdo da tarefa.

A figura 3.6 ilustra o funcionamento dessa estratégia. Na figura, uma nova *"task provider"*, *"task-1296"*, está sendo inserida em um sistema que possui 3 *"machine providers"* e 2 *"task providers"*. O padrão gráfico da *"machine provider"* (*"mach-0006"*) e da *"task-0792"* indica que a nova tarefa será inserida na mesma máquina física do provedor associado (Mendes et al., 2009).

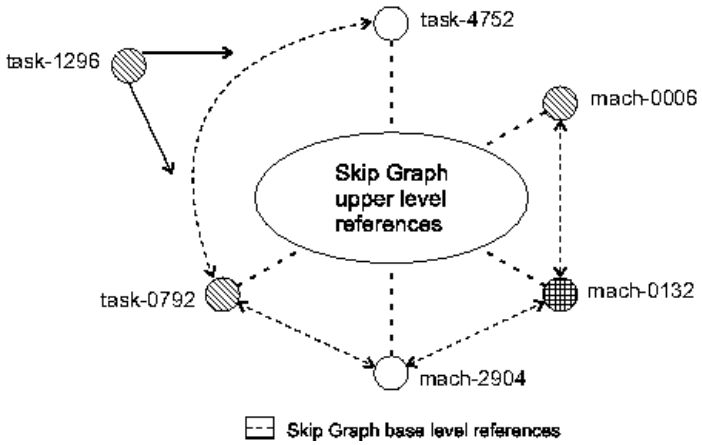


Figura 3.6: Um sistema com 3 *machine providers* e 3 *task providers*. (Mendes et al., 2009)

Nesse modelo, as *machine providers* e as *tasks providers* são escolhidas de maneira aleatória e incremental em um tempo $O(logn)$. A seleção incremental é realizada através da composição da rotina que obtém o primeiro *provider* de qualquer tipo (*machine* ou *task*) com a rotina de percorrer entre os nós vizinhos.

Para permitir a utilização da política de alocação *Self Scheduling*, sem que os nós contactem um nó central continuamente, como ocorre no modelo centralizado (seção 3.3) e, conseguir determinar o término da execução de todas as tarefas que compõem a aplicação, as *machines providers* simplesmente executam uma rotina constantemente em tempo $O(logn)$ e observam o seu retorno. Se o retorno for uma *task provider*, então a tarefa correspondente é executada, senão, se o retorno for $\perp$ então significa que todas as tarefas já foram executadas (Mendes et al., 2009).

Dessa forma, esse modelo consegue realizar a alocação de tarefas em ambientes distribuídos e descentralizado, e ainda assim, tomar conhecimento se todas as tarefas distribuídas já foram executadas, sem necessitar de um nó central.

3.5 Quadro Comparativo

A tabela 3.1 apresenta as principais características das estratégias de alocação de tarefas apresentadas nesta dissertação.

Quanto ao algoritmo, apenas a estratégia (Sousa and Melo, 2006) é centralizada, sendo que as estratégias (Repantis et al., 2005) e (Chakravarti et al., 2005) fazem uso de informações globais sobre o estado da topologia para tomar algumas decisões. As estratégias (Awan et al., 2006), (Bölöni et al., 2005) e (Mendes et al.,

2009) empregam algoritmos randômicos para alocar as tarefas. Nessas estratégias, a topologia da rede é representa através de estruturas de dados como árvore, rede *n-Cycle* e *Skip graph* (Aspnes and Shah, 2003). (Sousa and Melo, 2006) e (Chakravarti et al., 2005) implementam mais de uma política de alocação, inclusive a *Fixed* (Shao, 2001). Com exceção da estratégia (Bölöni et al., 2006), todas as demais são para alocação de tarefas do tipo *Bag-of-Task(BoT)*.

Na última linha do quadro ilustrado na tabela 3.1 são apresentadas as características do framework proposto na presente dissertação, que serão discutidas em detalhe no capítulo 4.

Nome	Estrutura	Tipo de Sistema	Política de Alocação	Tipo de Aplicação
(Repantis et al., 2005)	Hierárquico / Descentralizado	P2P	Fixed	BoT
(Chakravarti et al., 2005)	Hierárquico / Descentralizado	Grid	Adaptive Self-Scheduling, Preftech	BoT
(Awan et al., 2006)	Hierárquico / Descentralizado	P2P	Fixed	BoT
(Sousa and Melo, 2006)	Centralizado	Grid	Fixed, SS, GSS, TSS, FAC2 e Definida pelo usuário	BoT
(Bölöni et al., 2006)	Estruturado / *n-cycle*	P2P	Random Walk, W. St. Forward (WSF)	BoT
(Shen and Yuan, 2008)	Hierárquico / Descentralizado	P2P	*QoS* Based	BoT
(Mendes et al., 2009)	Estruturado / *Skip graph*	Grid	Self-Scheduliung	BoT
Framework proposto	Estruturado / Múltiplos	Grid	WSR, LWSR , Definida pelo usuário	BoT

Tabela 3.1: Quadro comparativo das estratégias de alocação de tarefas.

Capítulo 4

Projeto do Framework de Execução de Tarefas em Ambiente Distribuído e Descentralizado

O objetivo desse capítulo é apresentar o projeto de um framework flexível para execução de tarefas do tipo *Bag-of-Tasks* em ambiente distribuído e descentralizado.

A primeira seção apresenta as decisões de projeto que guiaram o desenvolvimento do framework. A segunda seção apresenta e descreve as camadas e componentes do framework. Por fim, a terceira seção realiza uma discussão sobre a flexibilidade do framework e das políticas de execução de tarefas.

4.1 Decisões de Projeto

O objetivo principal do framework proposto nessa dissertação é ser flexível o suficiente para que possa ser instanciado independente do *overlay* e do *middleware* de rede. A inclusão de novas políticas de execução de tarefas deve ser para o framework um *plugin*. O framework deve abstrair a complexidade inerente à arquitetura alvo, nesse caso, a arquitetura P2P, de modo que os desenvolvedores não tenham que lidar com complexidade de baixo nível, como formato das mensagens, roteamento, comunicação, entre outros. O framework proposto fornece um ambiente descentralizado para execução de tarefas *BoT* (seção 3.2.2) em ambiente distribuído e heterogêneo como *desktop grid*.

Para que o framework fosse independente do *overlay* e do *middleware* P2P definimos uma interface que abstrai o *overlay* utilizado de modo que podemos, por exemplo, utilizar a plataforma JXTA (JXTA, 2003) como *middleware* com o Chord (Stoica et al., 2001) como *overlay* P2P e *Work Stealing* (seção 3.3.5) para alocação das tarefas ou CAN (Ratnasamy et al., 2001) utilizando *Skip Graphs* (Aspnes and Shah, 2003) como *overlay* P2P estruturado para executar tarefas no modo FCFS (First-Come First-Served).

A estratégia de alocação de tarefas é implementada com *work stealing* e replicação. No modelo tradicional de *work stealing*, uma tarefa é "roubada" de um nó por outro que está ocioso (seção 3.3.5). Nesse momento, a tarefa é retirada da fila do nó original e colocada em execução no nó que estava ocioso. Esse comportamento é

adequado em ambientes homogêneos e imunes a falhas. No entanto, normalmente os ambientes são heterogêneos e suscetível à falhas. Dessa forma, decidimos por adaptar a política de *work stealing* às características dos sistemas distribuídos. Nesse modelo, quando um nó encontra-se ocioso, e uma política de *work stealing* está ativada, é realizada uma pesquisa por tarefas que estão aguardando execução em algum nó, e existindo, uma cópia da tarefa é entregue ao nó solicitante. Porém, essa tarefa não é retirada da fila do nó original, podendo essa mesma tarefa ser tomada por n outros nós. Uma tarefa será removida da fila apenas quando sua primeira execução for concluída. Essa decisão provê um certo nível de tolerância a falhas. Uma tarefa é considerada concluída a partir do instante que o submissor recebe a primeira notificação de execução da tarefa.

Além de optar-se por trabalhar com n réplicas, tomou-se a decisão de disponibilizar uma estrutura flexível que possibilite a adição de novas políticas de alocação de tarefas.

Por último, tendo como foco a definição e desenvolvimento de um framework extensível e flexível no que diz respeito as funcionalidades oferecidas, as mensagens foram definidas em um formato independente do protocolo utilizado. Nesse caso, o framework foi especificado com uma camada responsável por traduzir as mensagens de alto nível usadas pelas aplicações para mensagens de baixo nível e dependentes do protocolo utilizado.

4.2 Arquitetura do Framework Proposto

O framework proposto possui uma arquitetura modular, onde observa-se a existência de módulos especializados, totalmente desacoplados, para tratar determinadas ações. A figura 4.1 apresenta a organização de um nó que possui uma instância do framework. O framework proposto é dividido em três camadas: *Communication Layer*, *P2P Layer* e pela *Task Layer*. A camada *Task Layer* é formada pelos módulos: *Task Management Module* e *Notification Module*. O módulo *Task Management Module* é formado pelos submódulos *Task Allocation Policy* e *Task Executor*. As camadas do framework proposto e seus módulos e submódulos são detalhados na seções subseqüentes.

Essa arquitetura modular caracteriza o nosso framework para execução de tarefas. Nessa arquitetura, os módulos são desacoplados de modo a favorecer a troca e/ou inclusão de novos módulos específicos sem impactar os existentes. As tarefas são alocadas e executadas sem a presença de um nó centralizador, pois os nós tomam conhecimento um dos outros a partir do momento em que juntam-se à rede e o algoritmo de *join* da camada de infra-estrutura P2P é executado. Além disso, o controle sobre quando todas as tarefas foram executadas fica a cargo do nó submissor, através do tratamento dos eventos, e a ociosidade dos nós é em parte reduzida através das políticas de *work stealing* implementadas. Da mesma maneira, a presença de um módulo específico para a alocação de tarefas permite a implementação de diferentes estratégias de alocação. Por fim, a existência de um módulo de execução possibilita que o framework seja instanciado para diferentes tipos de aplicações, sendo o processamento específico de cada aplicação encapsulado nesse módulo.

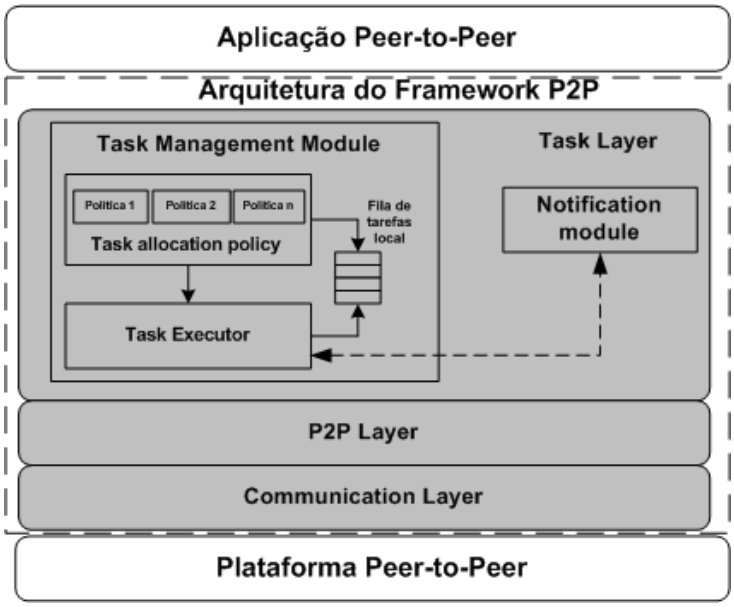


Figura 4.1: Estrutura da Aplicação P2P

4.2.1 Communication Layer

Um framework P2P deve ser suficientemente genérico para possibilitar o uso de diferentes tipos de protocolo/sistema. Por isso, é necessária uma camada responsável por traduzir chamadas de alto nível em outras de baixo nível, dependentes do protocolo/sistema empregado. Dessa forma, esta camada é responsável por fazer a interface com o protocolo/sistema P2P empregado, expondo interfaces mais simples para as camadas superiores sem expor qual o protocolo/sistema utilizado pela aplicação.

A Camada de Comunicação (*Communication Layer*) é responsável por abstrair o *middleware* de rede do restante do framework, provendo um formato de mensagem independente do *overlay* de rede utilizado.

Toda a comunicação da aplicação com a plataforma P2P é realizada pela camada de comunicação. Isso possibilita que as demais camadas trabalhem com um formato de mensagem independente da plataforma P2P e não precisem tomar conhecimento sobre como construí-la, ou de informações como, por exemplo, identificação (endereçamento) e ou localização dos nós. Ao invés disso, as outras camadas delegam essa responsabilidade à camada de comunicação que atua como um broker entre a aplicação e a plataforma P2P utilizada.

No framework proposto nessa dissertação, uma mensagem é uma instância da interface *Message* que possui o conteúdo da mensagem, o destinatário e o nó de origem da mensagem como atributos. A camada de notificação disponibiliza uma interface que atua como *proxy*, com as operações disponíveis para serem invocadas por outras camadas, tais como, *void send(message)* e *void notify(message)*.

Por *default*, o *middleware* de rede utilizado é o JXTA (JXTA, 2003) e toda mensagem é traduzida para o formato conhecido pelo JXTA. Para utilizar outro protocolo é necessário implementar essa interface e colocar o nome da implementação

no arquivo de propriedade do sistema, modificando a propriedade padrão que é a *default.communication.name*.

4.2.2 P2P Layer

Um framework P2P, para ser utilizado por diferentes tipos de aplicações distribuídas, deve prover um nível de abstração suficiente, para que possa ser utilizado por diferentes tipos de sistemas e protocolos. Para isso, é fundamental a definição de uma interface responsável pela comunicação com o sistema ou protocolo utilizado, sem atrelá-lo à aplicação.

O objetivo da camada *P2P Layer* é abstrair o ambiente P2P utilizado permitindo que o framework seja instanciado com diferentes tipos de *middlewares* e *overlays* P2P. Por exemplo, uma aplicação que instancie esse framework pode ser executada em uma estrutura que utilize *Skip Graph* (seção 2.5) ou Chord (seção 2.3) ou mesmo algum outro *overlay* definido pelo usuário. A flexibilidade e extensibilidade é alcançada por essa camada através da disponibilização de uma interface de alto-nível, com operações suficientemente genéricas que possibilitam os outros módulos serem executados de forma independente do *overlay* P2P. Essa flexibilidade possibilita a troca da implementação da infra-estrutura P2P sem impactar os demais componentes da arquitetura.

Dessa forma, o objetivo dessa camada é fornecer as funcionalidades de uma estrutura P2P, como *placement*, *search*, entre outras. A tabela 4.1 apresenta as operações disponibilizadas por essa camada.

Operação	**Descrição**
create	Cria a estrutura P2P
join	Realiza o *join* de um novo nó na estrutura P2P
insert	Insere um novo recurso com um identificador (*key*) em um nó n
registry	Registra os eventos os quais um nó esteja interessado
leave	Desconecta o nó da estrutura P2P

Tabela 4.1: Operações da Interface da Camada P2P.

A operação *create* é executada por um nó quando esse quer criar uma estrutura P2P, ou quando esse nó não tem conhecimento sobre a existência de outros nós, ou seja, de alguma estrutura P2P existente.

A operação de *join* possibilita aos nós juntarem-se à rede utilizando algum outro nó conhecido, pertencente à estrutura.

Nesse trabalho, para não exigir a figura do *bootstrap*, os nós podem executar a operação *create*, pois, mesmo que um nó crie uma nova estrutura P2P, esse nó ficará isolado em sua estrutura até a primeira execução da operação de descoberta de nós, que é executada quando o módulo detecta que o nó encontra-se "isolado" na rede. Para descobrir outros nós, o framework envia uma mensagem em *broadcast*, contendo o nome do nó, e o seu grupo. Os outros nós, ao receberem uma mensagem de descoberta comparam se o grupo do emissor é o mesmo do seu, e respondem com uma mensagem contendo o seu nome e a sua localização, para que o nó solicitante possa executar a operação de *join*.

A operação de *insert* possibilita aos nós inserirem recursos (tarefas) que são colocados na sua estrutura local. Todo recurso inserido possui um identificador (*key*) calculado a partir do seu conteúdo, utilizando algum algoritmo de *hashing*.

O framework disponibiliza a operação de *registry* para que os nós registrem os eventos nos quais estão interessados. Para o framework, um evento é a ocorrência de alguma ação que possa ser visível aos nós. Como exemplo de evento, temos: a inserção ou execução de uma tarefa, o *join* de um nó na estrutura, dentre outros. Para registrar um evento, o nó fornece uma instância do evento no qual esteja interessado, que segue a interface definida pelo framework.

Um nó pode se desconectar da estrutura executando a operação *leave* que, antes de desconectar, verifica se há tarefas na fila local do nó aguardando execução. Caso afirmativo, essas tarefas são transferidas para outro(s) nó(s), que são notificados para que atualizem a referência ao nó desconectado.

Além das operações apresentadas na tabela 4.1, a camada *P2P Layer* implementa internamente algumas operações de manutenção da infra-estrutura como as operações de: verificação de isolamento do nó, verificação da consistência da estrutura P2P. Essas operações são úteis especialmente para garantirem a consistência do *overlay*, pois os nós podem se desconectarem da estrutura sem executar a operação *leave*, na ocorrência de falhas, por exemplo, e por isso, são executadas continuamente em *background* com o intervalo entre as execuções definido via parâmetro de aplicação.

4.2.3 Implementação do Chord

Com objetivo de adotar um *overlay* decentralizado que possuísse um protocolo de estabilização eficiente, para que o desempenho do sistema não seja prejudicado em casos onde haja alta rotatividade de nós. Optamos por utilizar o Chord (seção 2.3) como *overlay* P2P, por provê descentralização, escalabilidade, disponibilidade e balanceamento de carga entre os nós participantes, além de um algoritmo eficiente para localização das chaves, mesmo em ambientes com elevado número de entrada e saída de nós da rede. Inicialmente *Skip Graphs* (seção 2.5) era a estrutura preferencial para ser utilizada na implementação do *overlay*, com os nós dispostos segundo a proposta de (Mendes et al., 2009). A escolha do *Chord* em oposição ao *Skip Graph* deu-se porque este não oferece um protocolo eficiente de estabilização dos nós, o que em um ambiente real pode levar a estrutura a um estado inconsistente ou a depreender tempo significativo na estabilização, prejudicando a inserção de novos nós ou na operação de busca na rede.

O *Chord* foi implementado como *overlay* P2P, com as chaves (*keys*) sendo calculadas a partir do conteúdo da tarefa, utilizando o algoritmo de *hash SHA-1*.

No *Chord* um nó precisa conhecer ao menos um nó, para juntar-se à rede. O nó conhecido é denominado *bootstrap* para realizar o *join* ou então o nó cria uma nova estrutura executando a operação *create*. Isso implica que existindo uma estrutura, o nó para juntar-se devem conhecer pelo menos um nó que pertença a essa estrutura. No entanto, nesse trabalho, como descrito na seção 4.2.2, optamos por oferecer aos nós a opção de juntarem a rede executando a operação de *join* ou a operação *create*, mas com um "serviço" de descoberta de nós.

A figura 4.2 apresenta a estrutura de classes da implementação do Chord(Stoica et al., 2001). O Chord implementa a interface exposta pela camada *P2P Layer*, explicada na seção 4.2.2 e toda a especificação descrita em (Stoica et al., 2001), incluindo as tarefas de manutenção da estrutura, representada pelas classes: *Discovery*, *FixReference*, *Stabilize* que são respectivamente, tarefas para descobrir outros nós que estejam desconectados da estrutura, verificar e corrigir as referências não válidas da *Finger Table* e por notificar os sucessores de um nó sobre a sua existência ou saída da rede. A *Finger Table* foi encapsulada através da classe *FingerTable* que internamente utiliza de uma estrutura de dados do tipo $<$*key, value*$>$(*Map*), onde a *key* é o identificador do nó ($N.id$) e o *value* a referência ao nó. As tarefas de manutenção são executadas continuamente com o intervalo de execução definido via parâmetro de aplicação, ou no caso das *FixReference* ou *Stabilize*, quando as operações de *join* ou *leave* são executadas.

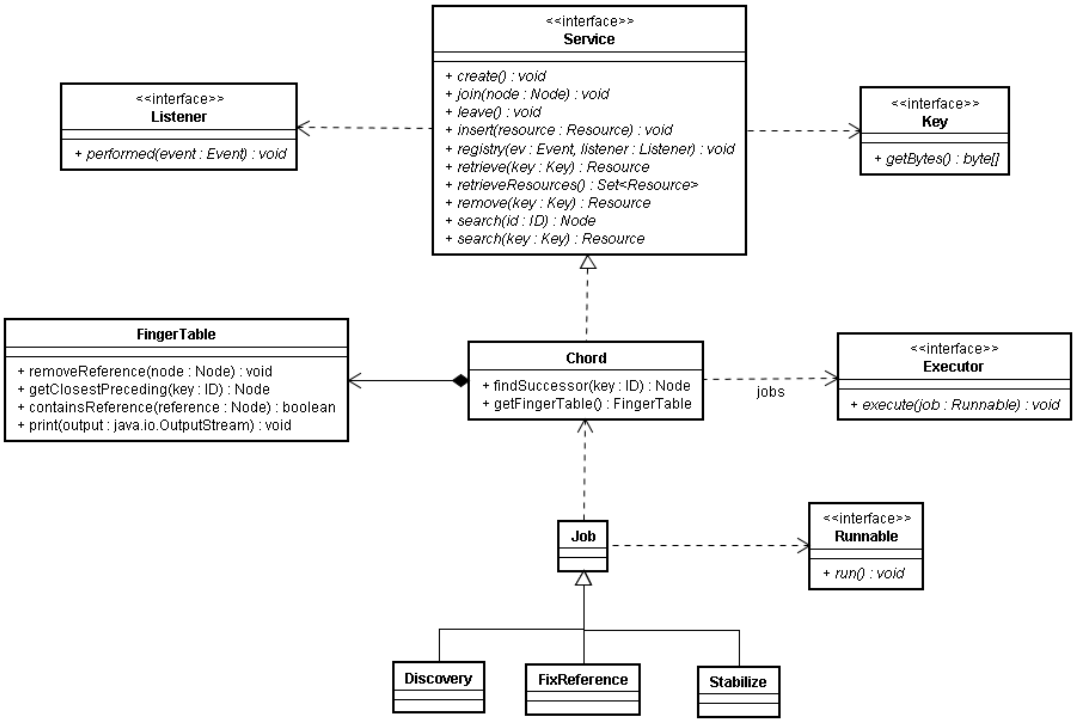


Figura 4.2: Estrutura de classes da Implementação do Chord.

4.2.4 Task Layer

A camada *Task Layer* é formada pelos módulos: *Task Management Module* e *Notification Module*. O *Task Management Module* é composto pelos sub-módulos: *Task Allocation Policy* e *Task Executor*, responsáveis respectivamente pela implementação das políticas de alocação e pela execução das tarefas na plataforma alvo. Esses sub-módulos se comunicam um com o outro através da fila local de tarefas (figura 4.1). A comunicação entre o módulo *Task Management Module* e o *Notification Module* ocorre através do sub-módulo *Task Executor*, que manda ao módulo de notificação os eventos que devem ser comunicados aos interessados, ou seja, os nós

que em algum momento executaram a operação *registry* passando a natureza do evento o qual estão interessados, conforme definido na tabela 4.1.

4.2.4.1 Task Allocation Policy

O Módulo de Alocação de Tarefas (*Task Allocation Policy*) é responsável por instanciar uma política de alocação de tarefa e alocar a tarefa no nó correspondente. A figura 4.3 apresenta a interface publicada por esse módulo.

A interface *TaskAllocationPolicy* define uma operação suficientemente genérica *void allocate(task)*, e as políticas de alocação são implementações dessa interface. Como *default*, foi implementada a política *PushBasedAllocate*, onde as tarefas são uniformemente distribuídas aos nós e colocadas para execução na fila local de cada nó usando a camada *P2P Layer* (seção 4.2.2). A adição de novas políticas é realizada através da implementação dessa interface e do registro da nova implementação no arquivo de propriedade do sistema, substituindo a política de alocação *default* pela nova política implementada. Ainda, pode-se informar ao framework que a implementação da política será definida em tempo de execução. Nesse caso, deve ser também fornecido o mecanismo de Injeção de Dependência (*Dependency Injection*). A idéia básica da Injeção de Dependência é ter um objeto separado, *assembler*, responsável por popular um atributo em um objeto com a implementação apropriada para a interface. Nesse caso, a classe de implementação da interface não é ligada ao framework em tempo de compilação, mas como um *plugin*.

O arquivo de propriedade é organizado como um conjunto de entradas compostas pelo par <*chave*,*valor*>, onde a *chave* é o nome de uma propriedade única do sistema, nesse caso, o nome da política de alocação e o *valor* é o nome da classe completa que implementa a interface *TaskAllocationPolicy*.

No framework proposto nessa dissertação, a submissão e alocação de tarefas seguem as seguintes regras:

- Todo nó aceita as tarefas enviadas a ele para execução, independente de ter ou não recurso disponível no momento para executá-las.
- O nó onde a tarefa é alocada deve ser notificado, utilizando-se dos eventos.
- Toda tarefa possui duas ligações:
  1. Uma para o nó que a submeteu (sentido da informação).
  2. Outra para o nó no qual está alocada.
- Todo nó, quando não possui tarefa para executar, e uma política de *work stealing* está habilitada, dispara uma consulta por tarefa que esteja aguardando execução em algum nó. Caso exista tal tarefa, o nó obtém uma cópia da tarefa, executa-a, e ao final informa o término da execução ao nó submissor e ao nó da alocação inicial.

Por padrão, o framework utiliza a chave: *default.task.allocation.name* ou utiliza da estratégia de *IoC* implementada, que procura por uma implementação da interface de alocação para instanciar. Essa forma possibilita que novas implementações sejam adicionadas e utilizadas pelas aplicações de forma desacoplada.

A interface do módulo *Task Allocation Policy* é apresentada na figura 4.3.

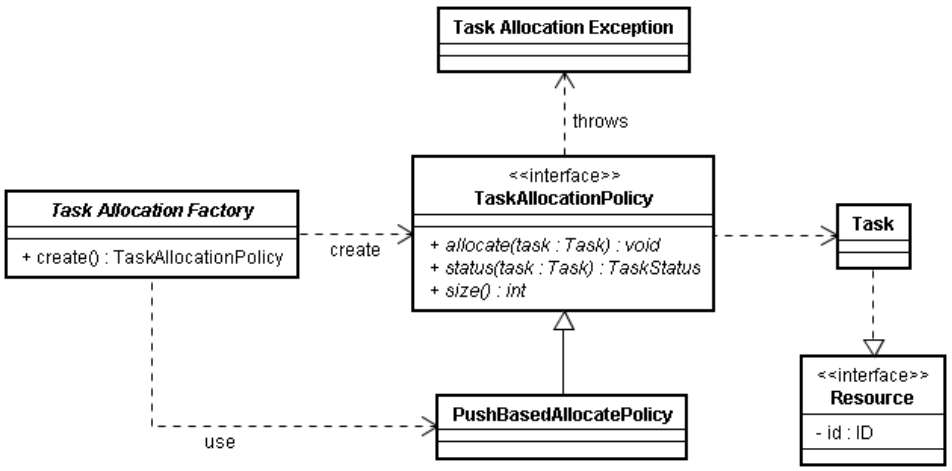


Figura 4.3: Interface do Módulo de Alocação de Tarefa

O retorno da operação de alocação é ***void*** e caso ocorra algum erro na alocação da tarefa, o nó que submeteu a tarefa é notificado através da exceção *TaskAllocationException*.

O algoritmo 1 apresenta a seqüência executada na alocação da tarefa em um nó.

Algoritmo 1: Task Allocate(t)

1 $n' \leftarrow n$.allocate(t)
2 t.allocators $\cup$ {n'}
3 n'.q $\cup$ {t}

No algoritmo 1 n é o nó submissor da tarefa e n' o nó onde a tarefa foi alocada. Na linha 1 o algoritmo executa a operação de alocação de tarefa a partir do nó submissor (n). Essa responsabilidade é delegada à camada *P2P Layer*. Quando a tarefa é inserida na fila do nó, atualizam-se os dados da tarefa, colocando a referência ao nó da alocação. A decisão de guardar a informação sobre os nós que alocaram a tarefa é necessária pois, quando a tarefa for "roubada" por outro nó, os nós nos quais a tarefa foi alocada, devem ser notificado(s) sobre a sua execução.

Além da operação de alocação, esse módulo disponibiliza as operações: *size()* e *status*, apresentadas na figura 4.3, que possibilitam obter a quantidade de tarefas que estão na fila local do nó aguardando execução, e o estado de uma tarefa que esteja na fila ou que já tenha sido executada e retirada da fila local do nó. Além disso, fica a cargo desse criar a ligação (*link*) entre o nó executor e o nó que submeteu a tarefa e com os demais nós interessados em obter informações sobre a tarefa.

4.2.4.2 Task Executor

Esse módulo é responsável por implementar a execução das tarefas. É através desse módulo que as tarefas são executadas na máquina alvo ou retiradas de execução, quando solicitado. O módulo de execução de tarefas (*Task Executor*) controla

todo o processo de execução da tarefa na plataforma alvo, desde o seu início até o seu término, isolando os detalhes referentes à execução da tarefa na máquina alvo. Para isso, o módulo fornece uma interface simples que deve ser implementada conforme a natureza de execução da tarefa, utilizando os parâmetros necessários à sua execução.

Quando uma tarefa é colocada na fila do nó, através do módulo de alocação de tarefa, este notifica o módulo de execução, que executa as tarefas na ordem FIFO, uma de cada vez, até a fila ficar vazia.

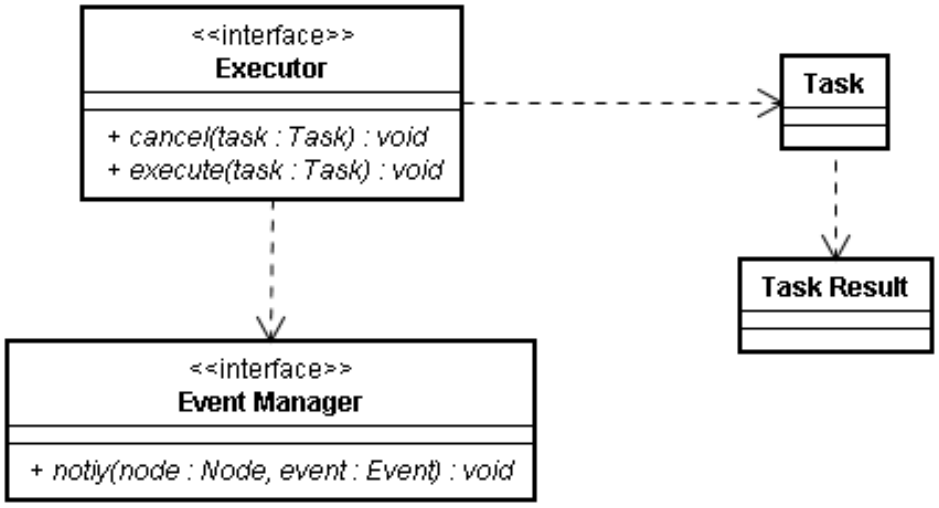


Figura 4.4: Interface do Módulo de Execução de Tarefa

Na figura 4.4, temos a definição das operações disponíveis na interface de execução. Nessa interface, a operação *execute* recebe a tarefa que deve iniciar a execução e a operação *cancel* cancela a execução da tarefa. De maneira geral, o trabalho realizado pelo módulo de execução é o descrito no algoritmo 2.

Algoritmo 2: Task Executor(task)

```
1  notify(task.allocators, start)
2  notify(task.owner, start)
3  if task.numberOfExecutions = max then
4      notify(task.allocators, error)
5  end
6  task result ← execute(task)
7  if task result fail then
8      inc(task.numberOfExecutions)
9      q ← ∪ {task}
10 else
11     notify(task.owner, finish)
12     notify(task.allocators, finish)
13 end
```

Ao iniciar a execução de uma tarefa, o módulo de execução de tarefas notifica o(s) nó(s) que possui(em) réplica da tarefa, incluindo o responsável pela submis-

são (*task owner*), através do módulo de notificação, sobre o início da execução da tarefa. O tratamento desse evento fica a cargo de cada nó notificado. Quando a execução da tarefa apresenta erros, a tarefa é adicionada ao final da fila para uma nova tentativa, até que o limite máximo especificado seja alcançado. Nesse caso, se nenhuma das tentativas de execução obter sucesso, a tarefa é removida da fila de execução do nó e os nós interessados são notificados sobre a falha. Esse comportamento possibilita a aplicação tratar o erro conforme a sua necessidade, ou até mesmo, submeter a tarefa novamente para execução, conforme mostra a figura 4.5.

Ao ser criada, a tarefa é colocada no estado *"created"*. O nó ao submeter uma tarefa, o estado da tarefa muda para *"submitted"* e para *"allocated"* quando a tarefa é inserida na fila local de um nó. Ao entrar em execução, o estado da tarefa muda para *"executing"*. Caso a tarefa seja executada sem erros, o seu estado muda para *"executed"*. Caso contrário, o estado da tarefa muda para *"error"* e se o máximo de re-execução da tarefa ainda não estiver atingido o limite configurado, a tarefa é inserida novamente na fila local do nó e o seu estado muda para *"allocated"*.

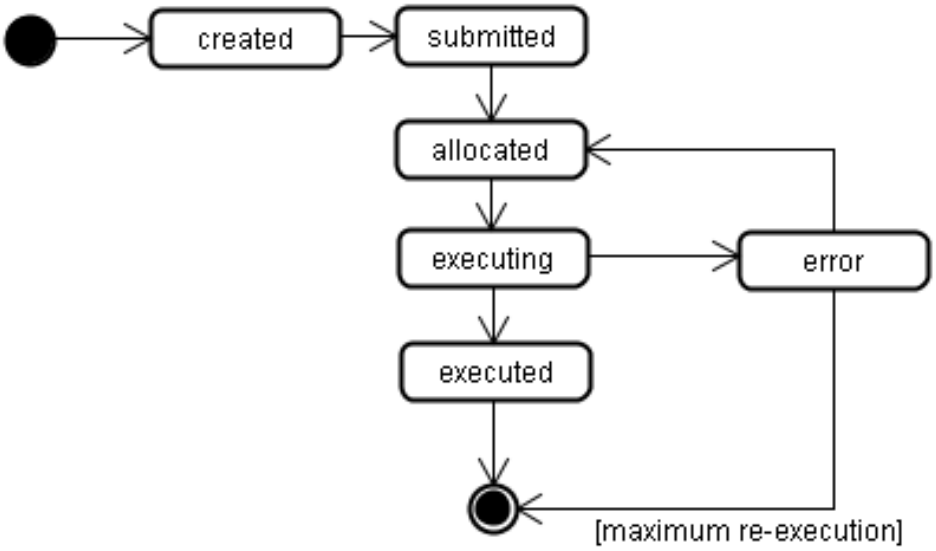


Figura 4.5: Estados válidos de uma Tarefa

A comunicação entre o módulo de alocação(*Task Allocation Policy*) e o módulo de execução (*Task Executor*) ocorre através de notificações (eventos) que são publicados sempre que uma tarefa é adicionada à fila de execução, ou quando a fila fica vazia, seguindo o conceito de produtor / consumidor. O módulo de Alocação atua como produtor de tarefas a serem "consumidas" pelo módulo de Execução, que fica bloqueado quando não há tarefas na fila, e é desbloqueado no instante que a tarefa é inserida na fila de execução e o módulo é notificado. Quando a fila de execução de tarefas fica vazia, o módulo de execução entra no estado de bloqueado e ativa a política de alocação de tarefas.

4.2.4.3 Notification Module

O módulo de notificação (*Notification Module*) é responsável por gerenciar e propagar os eventos, que são utilizados para notificar os outros módulos sobre alguma mudança no ambiente, que pode ser, por exemplo, a inserção de uma nova tarefa, a disponibilidade ou indisponibilidade de um nó ou mudança de estado de uma tarefa.

Esse módulo possibilita que as notificações sejam realizadas à medida que os eventos acontecem. Através desse módulo, os nós da rede podem notificar uns aos outros, porém, sem ter conhecimento da sua localidade ou até mesmo, da sua existência, uma vez que os nós não sabem quem está registrado como interessado em um evento, ou seja, os nós não sabem que nós eles notificam. Por intermédio desse módulo, é possível saber o instante em que uma tarefa foi executada, ou até mesmo finalizada, sem que seja necessário realizar uma operação tradicional de consulta na rede.

Um evento possui informação sobre o nó que deve ser notificado e sobre o que está sendo notificado. Todo nó pode se registrar, através da operação *registry* (tabela 4.1), como interessado em algum evento. Essa operação tem como parâmetro o evento *ev* que deve ser registrado e a operação que deve ser executada quando o evento ocorrer, através dos *Listeners*. Um *Listener* é uma interface que disponibiliza a operação *void performed (Event)* que possui implementação dependente da aplicação.

O framework já disponibiliza alguns *listeners* para os seguintes eventos:

1. Inserção de tarefa: Todo nó pode se registrar como interessado em receber notificações sobre a inserção de tarefas. Quando uma tarefa é alocada em um nó, o nó que recebeu a tarefa é notificado a respeito, e também, todos os nós que se registraram, executando a implementação do *listener* que foi informado por cada nó, no instante do registro.

2. Saída de um nó da estrutura: Quando um nó sai da estrutura executando a função *leave* (Tabela 4.1), alguns nós, obtidos através da camada *P2P Layer*, são notificados para que atualizem as suas referências. Se, durante a execução da rotina de manutenção do *overlay* P2P utilizado, for detectada alguma referência inválida, os nós são notificados para que removam a informação sobre essa referência inválida.

3. Isolamento na estrutura: Um nó, ao perder conectividade com os demais nós da rede, pode ser notificado para que execute a operação de descoberta de nós. Isso não ocorre apenas quando o nó perde conectividade, mas também quando o nó cria o *overlay*, executando a função *create* (Tabela 4.1) ao invés de um *join* e já existem outros nós na rede formando um *overlay* para execução de tarefas.

4.2.5 Políticas de Work Stealing

Na política de *work stealing*, quando um nó fica ocioso, o seu módulo de alocação de tarefas dispara uma consulta por tarefa, que será obtida de algum nó que possua tarefa aguardando execução na sua fila. Ao contrário da proposta tradicional de

work stealing (seção 3.3.5), nossas estratégias não excluem a tarefa da fila original, mas sim, entregam uma cópia da tarefa ao nó solicitante. Essa estratégia foi adotada para prover um nível de tolerância a falhas.

Ao final da execução de uma tarefa, o nó que a submeteu é notificado, assim como todos os nós que possuem uma cópia da tarefa em suas filas. Se, durante a execução de uma tarefa, um nó receber uma notificação de conclusão da tarefa por outro, a execução da tarefa é imediatamente cancelada e o nó inicia a execução da próxima tarefa da sua fila.

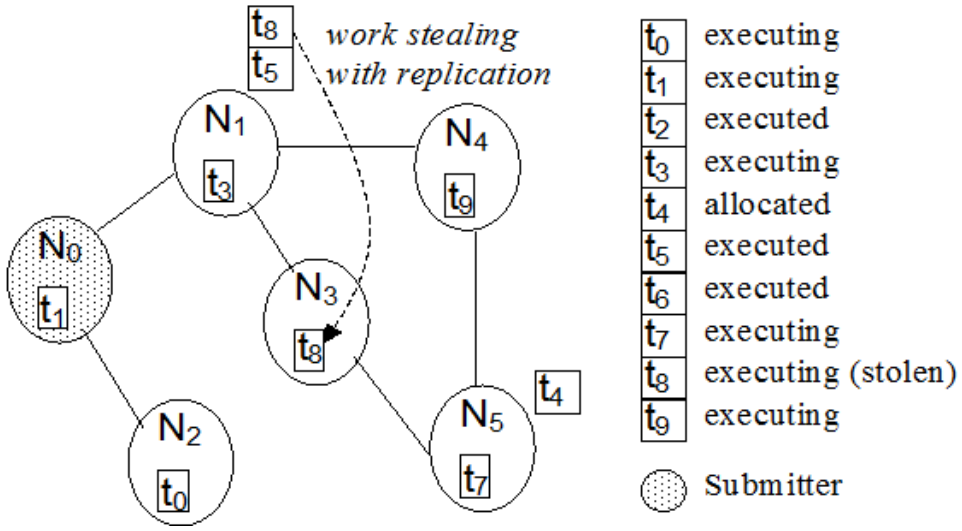


Figura 4.6: Interface do Módulo de Alocação de Tarefa

A figura 4.6 apresenta um sistema com 6 nós (N_0 a N_5) que estão executando 10 tarefas (T_0 a N_9). Quando o nó N_3 fica ocioso, ele rouba a tarefa t_8 do nó N_1. A escolha da tarefa a ser entregue ao nó é realizada de forma aleatória, para diminuir a chance de dois nós obterem a mesma tarefa em consultas sucessivas. Nesse caso, a tarefa t_8 é executada pelo nó N_3 e ao final da execução, os nós N_0 e N_1 são notificados. O nó N_1 remove t_8 da sua fila e o estado de t_8 muda para executada(*executed*).

Até o momento, duas políticas de *work stealing* foram integradas ao módulo de alocação:

1. *Local Work with Replication (LWSR)* - Política de *work stealing* que consulta apenas os nós que estão a um salto(*hop*) de distância do nó ocioso.
2. *Random Work with Replication (WSR)* - Política de *work stealing* que considera todos os nós na pesquisa por tarefas que estejam aguardando execução.

Como pode ser observado, a diferença entre as políticas implementadas restringe-se apenas ao conjunto de nós que podem ser incluídos na consulta aleatória por tarefas. Na política *Local Work with Replication (LWSR)*, os nós a serem consultados sobre a existência de tarefas aguardando execução são aqueles que estão a um salto de distância do nó ocioso. Na política *Random Work with Replication (WSR)*, a escolha do nó a ser consultado é realizada de forma aleatória considerando todos os nós do sistema. Na figura 4.6, quando o nó N_3 fica ocioso, a política WSR escolhe

aleatoriamente um nó entre $\{N_0, N_1, N_2, N_4, N_5\}$ para ser consultado, enquanto que a política *LWSR* considera apenas os nós $\{N_1, N_7\}$.

No algoritmo 3, temos o procedimento executado pela política LWSR e no algoritmo 4, o da política WSR.

Algoritmo 3: Local work stealing

```
1 n1 ← n.find(n.id)
2 task ← n1.askForTask()
3 if task <> null then
4     q ∪ {task};
5 end
```

Algoritmo 4: Random Work Stealing

```
1 k ← random()
2 n1 ← n.find(k)
3 if n1 <> n then
4     task ← n1.askForTask()
5     if task <> null then
6         q ∪ {task};
7     end
8 end
```

Nos algoritmos 3 e 4, n é o nó ocioso, n_1 é o nó que será consultado e q a fila de tarefas do nó n.

Capítulo 5

Resultados Experimentais

O objetivo desse capítulo é descrever e discutir os resultados obtidos na utilização do *framework* proposto no capítulo 4 e das políticas de *work stealing* com replicação para a execução de uma aplicação distribuída de balanceamento de tráfego aéreo.

A primeira seção descreve o ambiente de testes, incluindo as características do hardware e software utilizados. A segunda seção apresenta detalhes sobre as aplicações distribuídas utilizadas nos experimentos. Por fim, a terceira seção apresenta os resultados obtidos na execução das aplicações.

5.1 Ambiente de Execução

Para os testes, foi utilizado um *desktop grid* de 16 máquinas heterogêneas conectadas pela rede do Departamento de Ciência da Computação da Universidade de Brasília (UnB), dos quais cinco são Intel Pentium *Core 2 Duo* e sete são *AMD Sempron* de 1.3 GHz, e um *notebook* Pentium Dual. A tabela 5.1 apresenta as características de cada máquina.

Cada máquina utiliza o Linux ou o Windows Vista como sistema operacional. Um protótipo do framework foi implementado em Java e JXTA. As aplicações de testes foram compiladas utilizando o Java versão 1.6.0_14, utilizando o MySQL 5.0 como banco de dados e o JXTA-J2SE 2.5 como *middleware* P2P. A tabela 5.2 relaciona os software utilizados.

Além disso, foi utilizada a implementação do Chord descristo na seção 4.2.3 com intervalo de 20 segundos para execução das tarefas de manutenção (*Discovery*, *FixReference* e *Stabilize*).

As 16 máquinas utilizadas nos experimentos estão dispostas em dois laboratórios distintos, Labpós e Translab, conforme mostra a figura 5.1. As máquinas nb, tl-01, tl-02, tl-03, tl-04, tl-05 e tl-06 ficam no Translab e as máquinas p-03, p-04, p-05, p-06, p-07, p-10, p-11, p-12 e p-13 ficam no Labpós. As máquinas de um laboratório se comunicam com as máquinas do outro através de um *switch* de 100 Mbps.

	Processador	Memória	HD
NB	1.6 GHz	2 GB	120 GB
TL01-03	2.66 GHz	4 GB	300 GB
TL04-06	1.6 GHz	2 GB	140 GB
P03	2.53 GHz	2 GB	150 GB
P04-07, P10-13	1.3 GHz	2 GB	40 GB

Tabela 5.1: Características dos computadores utilizados.

Nó	**Versão**	**Descrição**
Debian	2.6.26	Sistema Operacional
Windows	Windows Vista Service Pack 2	Sistema Operacional
Java JDK	1.6.0_14	Compilador e JVM
MySQL	5.0	Banco de dados

Tabela 5.2: Características dos softwares utilizados.

5.1.1 Aplicação Distribuída

Para avaliar o framework proposto e as políticas de *work stealing* com replicação era preciso implementar uma aplicação distribuída do tipo *Bag-of-Tasks*. Para isso, três aplicações de balanceamento de tráfego aéreo foram projetadas e desenvolvidas utilizando os algoritmos de fluxo máximo apresentados na seção I.4. Os algoritmos implementados foram: Edmonds-Karp (Edmonds and Karp, 1972), Dinic (Dinic, 1970) e FIFO Preflow Push (Ahuja and Orlin, 1989).

Problemas de fluxo de rede vem sendo estudados há décadas e, apesar de diversos algoritmos terem sido propostos na literatura, sabe-se que a escolha de um algoritmo particular depende do tipo de problema abordado e do tempo de que se dispõe para resolvê-lo.

Para decidir o melhor algoritmo para um determinado cenário usualmente são feitas simulações de diversos algoritmos, o que possibilita escolher aquele que apresente o melhor resultado. Devido ao tamanho das redes de fluxo simuladas, geralmente as simulações são executadas em ambientes distribuídos, onde cada máquina fica responsável pela simulação de um subconjunto de grafos (cenários) em um subconjunto de algoritmos.

Para execução dos algoritmos era necessária uma base de dados com os movimentos aéreos a serem balanceados. Nos nossos experimentos foram utilizados os dados dos movimentos aéreos dos dias 30 de abril e 02 de maio de 2008 disponibilizados pelo Centro Integrado de Defesa Aérea e Controle de Tráfego Aéreo (CINDACTA I) e inseridos no banco de dados MySQL.

Dessa forma, as tarefas da aplicação distribuída executam um dos algoritmos de fluxo máximo utilizando os dados dos movimentos para formar o grafo de movimentação das aeronaves, que seriam utilizados como entrada para os algoritmos.

5.1.2 Preparação dos experimentos

O planejamento para execução dos experimentos contou com as seguintes fases:

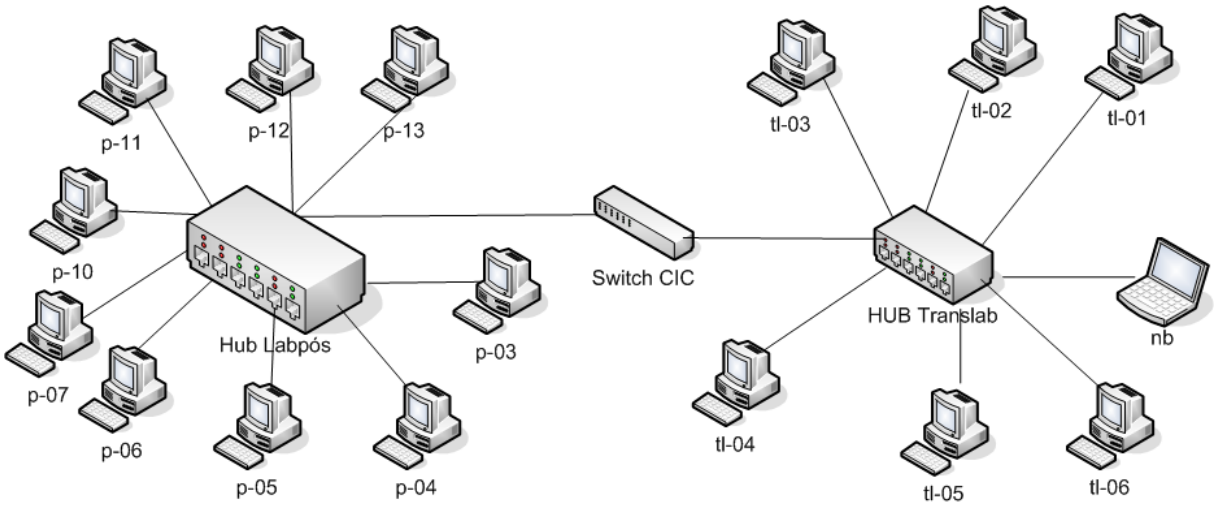


Figura 5.1: Disposição das máquinas utilizadas nos testes.

1. Definição e implementação da aplicação distribuída.
2. Implementação do *overlay* P2P.
3. Definição da quantidade de nós para executar a aplicação distribuída.
4. Definição do número de tarefas de cada execução.
5. Execução da aplicação de forma seqüencial no nó de melhor, pior e com média capacidade de processamento.
6. Execução da aplicação com as políticas de *work stealing* habilitadas (WSR, LWSR) e a política sem *work stealing* (NoWS).

As aplicações distribuídas foram executadas em 1, 2, 4, 8 e 16 nós. A tabela 5.3 apresenta os nós utilizados em cada execução.

Número de Nós	**Nó**
1	tl-01
2	nb, p-11
4	nb, p-04, p-05, p-12
8	nb, p04-06, p10-12, tl-01
16	nb, tl01-06, p03-07, p10-13

Tabela 5.3: Nós utilizados em cada execução.

Na tabela 5.3, a máquina com maior capacidade de processamento foi escolhida para a execução seqüencial e as configurações 2, 4, 8 e 16 foram escolhidas de maneira a sempre conterem uma máquina com baixa capacidade de processamento.

Para os testes, foi definido que as aplicações *BoT* são compostas de 90, 180 e 270 tarefas. Cada tarefa executa um dos algoritmos de fluxo máximo implementados utilizando um grafo que expressa os movimentos aéreos das aeronaves no espaço aéreo brasileiro.

5.2 Experimentos

Para efeito de comparação do ganho obtido com cada uma das políticas, as máquinas tl-01, p-04 e p-05 foram escolhidas por serem respectivamente as que possuem, a melhor, pior e média capacidade de processamento. A tabela 5.4 apresenta o tempo de execução seqüencial da aplicação com 90, 180 e 270 tarefas em cada uma dessas máquinas.

Nome	**90 tarefas**	**180 tarefas**	**270 tarefas**
tl-01	627	1341	1951
p-04	903	1780	2668
p-05	893	1783	2634

Tabela 5.4: Tempo de execução seqüencial em segundos das máquinas: tl-01, p-04, p-05 e p-11.

Como esperado, a máquina tl-01 foi a que apresentou o melhor tempo de execução das aplicações, com uma diferença de 13,67% em relação ao tempo de execução da pior máquina para 270 tarefas.

Para avaliar as políticas de *work stealing* propostas, foram armazenados dados sobre o tempo de execução da tarefa, o nome do nó que submeteu a tarefa, ocorrência de erro na execução, juntamente com o número de tentativas de re-execução e se tarefa foi executada por um nó diferente daquele o qual havia sido inicialmente alocada, o que caracteriza um "roubo". Para isso, cada aplicação foi executada com um tipo de política habilitada. A política *NoWS* indica que nenhuma política de *work stealing* está habilitada, o que significa que cada nó executa apenas as tarefas que lhe foram alocadas.

As políticas WSR e LWSR são as políticas *Work Stealing With Replication* e Local *Work Stealing With Replication*, explicadas na seção 4.2.5. A tabela 5.5 apresenta o tempo de execução para 90, 180 e 270 tarefas com 2, 4, 8 e 16 nós, utilizando as políticas WSR, LWSR e NoWS, juntamente com o número de roubos efetuados em cada execução. A figura 5.2 apresenta o gráfico com os tempos de execução das aplicações em 2, 4, 8 e 16 nós com 90, 180 e 270 tarefas das políticas WSR, LWSR e NoWS.

Como pode ser observado, o tempo de execução da política NoWS foi o mais alto em todos os casos, com exceção do tempo de execução em quatro nós e 180 tarefas onde o tempo foi menor que o da política LWSR. Tal comportamento se justifica porque na política LWSR é adicionado um *overhead* no tratamento das solicitações de pedido por tarefas. Nesse caso, como a quantidade de nós participantes é pequena, o tempo imputado no sistema com o tratamento de eventos e entrega das cópias das tarefas fez com esse o tempo total de execução da política LWSR ficasse maior que o da política NoWS. Apesar da política WSR também possuir o mesmo *overhead*, houve mais roubos de tarefas com essa política (Tabela 5.5), o que possivelmente contribuiu para o balanceamento de carga, reduzindo o tempo de execução.

Em todos os casos, a política WSR apresentou melhor tempo de execução que a política LWSR. Esse comportamento acontece devido ao fato das requisições por tarefas na política LWSR ficarem concentradas. Por essa razão, existem situações

2 nós	**Tempo (s) / roubos**		
Número de Tarefas	**90**	**180**	**270**
WSR	350s/011	583s/012	869s/043
LWSR	361s/008	663s/040	889s/029
NoWS	382s/000	635s/000	892s/000
4 nós	**Tempo (s) / roubos**		
Número de Tarefas	**90**	**180**	**270**
WSR	273s/041	452s/116	636s/149
LWSR	303s/020	663s/082	664s/094
NoWS	313s/000	480s/000	712s/000
8 nós	**Tempo (s) / roubos**		
Número de Tarefas	**90**	**180**	**270**
WSR	152s/034	264s/060	340s/118
LWSR	200s/016	352s/041	429s/052
NoWS	220s/000	441s/000	577s/000
16 nós	**Tempo (s) / roubos**		
Número de Tarefas	**90**	**180**	**270**
WSR	079s/048	113s/099	198s/064
LWSR	106s/030	124s/075	212s/034
NoWS	129s/000	140s/000	239s/000

Tabela 5.5: Tempo de execução/número de roubos de tarefas para as políticas WSR, LWSR e NoWS.

onde um nó ocioso não conseguirá obter mais tarefas para execução, mesmo existindo tarefas a serem executadas no sistema.

A figura 5.3 ilustra uma situação desse tipo, onde existem 4 nós dos quais três (tl-01, tl-02, tl-04) estão com as suas filas vazias. Ao ser utilizada a política LWSR, o nó tl-01 irá realizar requisições por tarefas aos nós que estão a um salto (*hop*) de distância, nesse caso, os nós tl-02, e tl-04. Como os nós tl-02 e tl-04 também estão ociosos, ou seja, sem tarefas nas suas filas, o nó tl-01 não executa mais tarefas, pois os nós tl-02, e tl-04 roubam apenas uma tarefa por vez, do nó tl-03. A tarefa roubada pelos nós tl-02, e tl-04 não é inserida na fila local dos nós, mas são colocadas diretamente em execução através do módulo *Task Allocation Policy* (seção 4.2.4.1. Na política WSR tal situação não ocorre, já que a escolha do nó a ser consultado é global.

Na tabela 5.5, observa-se também que o número de roubos de tarefas na política WSR aumenta a medida que aumentamos o número de tarefas, com exceção na execução de 180 e 270 tarefas em 16 nós. Nesse caso, o número de tarefas roubadas diminuiu com 270 tarefas. Esse não foi o comportamento esperado, uma vez que nessa política não existe o problema de localidade que ocorre na política LWSR.

A tabela 5.6 apresenta o ganho da política WSR calculado em relação à política NoWS. O melhor desempenho foi obtido ao executarmos 180 e 270 tarefas em 8 nós, que foi respectivamente de $67,05\%$ e $69,05\%$.

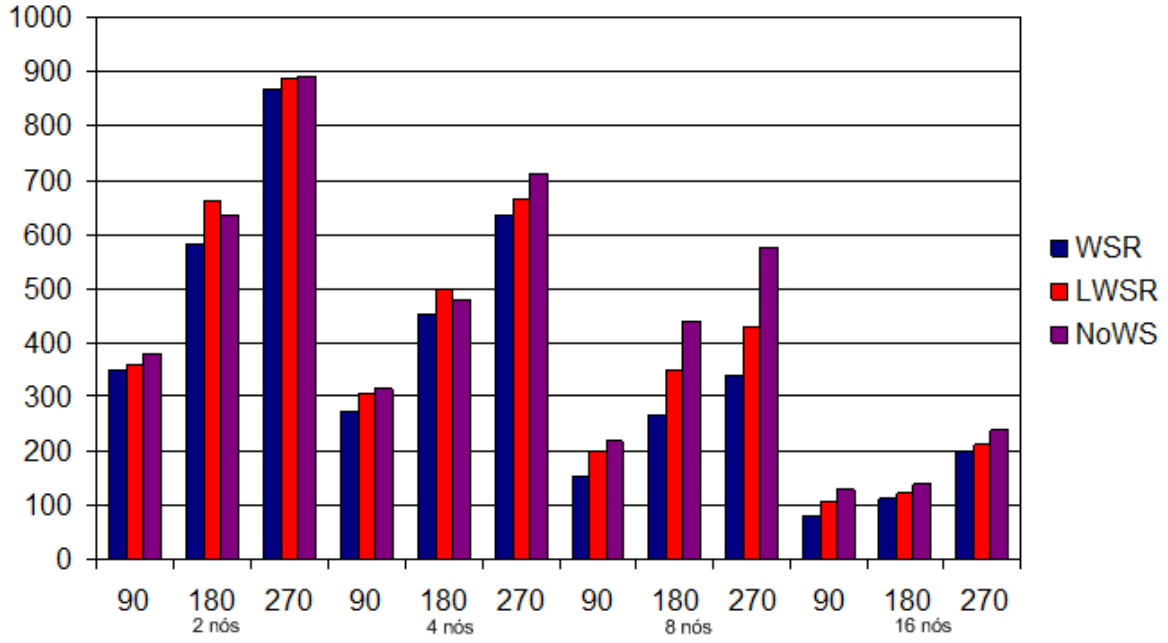


Figura 5.2: Tempo de execução para 2, 4, 8 e 16 nós com 90, 180 e 270 tarefas utilizando as políticas WSR, LWSR e NoWS.

Número de tarefas	2 nós	4 nós	8 nós	16 nós
90	9,14%	14,65%	44,74%	63,29%
180	8,92%	6,19%	67,05%	23,89%
270	2,65%	11,95%	69,71%	20,71%

Tabela 5.6: Ganho da política WSR sobre a política NoWS.

5.3 Speedup

A figura 5.4 apresenta o *speedup* obtido na execução de 90, 180 e 270 tarefas em 2, 4, 8, e 16 nós com a política WSR. Tomou-se como base o tempo de execução seqüencial da máquina mais rápida. Na execução com dois nós tivemos um *speedup superlinear*(2;3,5) e com 4, e 8 nós o melhor *speedup* foi obtido com 270 tarefas. No entanto, com 16 nós o *speedup* de 180 tarefas foi maior do que com 270 tarefas. O melhor *speedup* obtido, considerando todos os casos foi de $11,86$ com 16 nós e 180 tarefas.

Os *speedups* obtidos foram bastante satisfatórios uma vez que esse *speedup* foi calculado com base no tempo do melhor nó da rede, e os testes foram realizados em um ambiente heterogêneo e utilizando dois laboratórios do departamento. No caso de 16 nós e 180 tarefas, o tempo de execução foi reduzido de $22,35$ minutos para $1,8$ minutos.

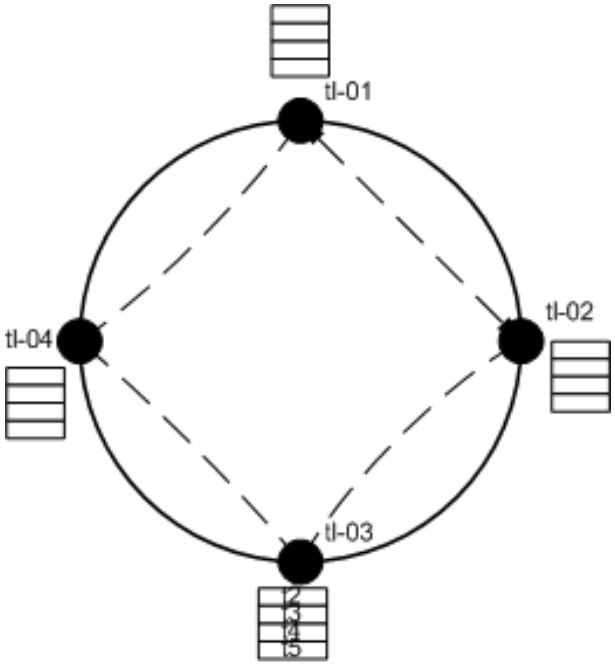


Figura 5.3: Exemplo de situação onde a requisição por tarefas de um nó não será será bem sucedida ao utilizarmos a política LWSR.

5.4 Ferramenta de Monitoramento de Execução

Para acompanhar a execução das tarefas foi desenvolvida uma ferramenta de monitoramento que fornece informações como: número e quais tarefas já foram executadas, as tarefas que estão aguardando execução, o tempo gasto na execução de cada tarefa, o número de tarefas roubadas por cada nó e o tempo total gasto na execução da aplicação.

Para o monitoramento da execução, a ferramenta registra-se como interessada nos eventos relacionados a mudança de estado das tarefas, que são disparados pelos nós toda vez que o estado da tarefa muda, através do módulo de notificação apresentado na seção 4.2.4.3, ou então consulta periodicamente o nó, sobre a situação das tarefas submetidas para execução.

Ao receber uma notificação, a ferramenta de monitoramento apresenta os dados sobre a tarefa que são: o endereço do *submitter*, a hora do nó submissor (*local time*), a hora do início da execução da tarefa (*start execution time*) em nanosegundos, o tempo de finalização da tarefa (finished *execution time*) em nanosegundos, o nó que executou a tarefa (*executor node*), o tempo de execução (*time*) da tarefa em segundos, se a tarefa foi roubada (*stealing*) e a informação sobre o nó que a tarefa foi alocada inicialmente (*responsible node*).

A figura 5.5 apresenta a ferramenta de monitoramento. Nesse exemplo, as tarefas *BoT* foram submetidas pelo nó (*164.41.14.35*) e os nós tl-03 (*164.41.14.62*), tl-04 (*164.41.14.25*),p-03 (*164.41.14.43*) conseguiram roubar e executar tarefas no sistema.

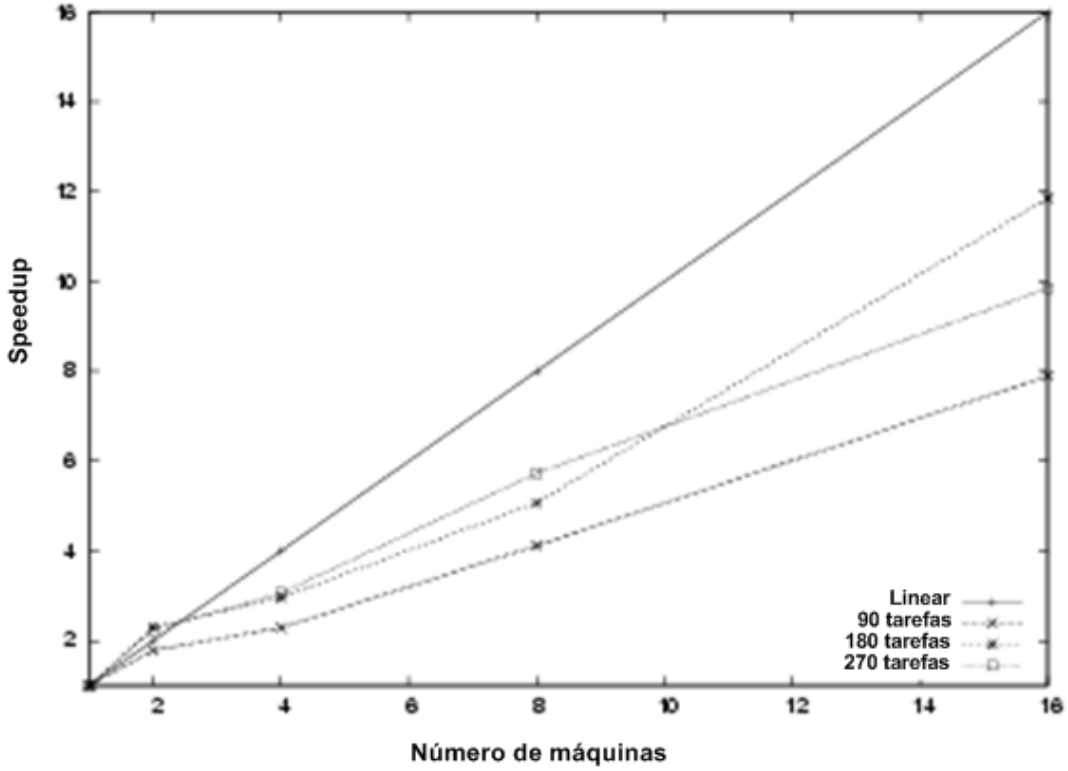
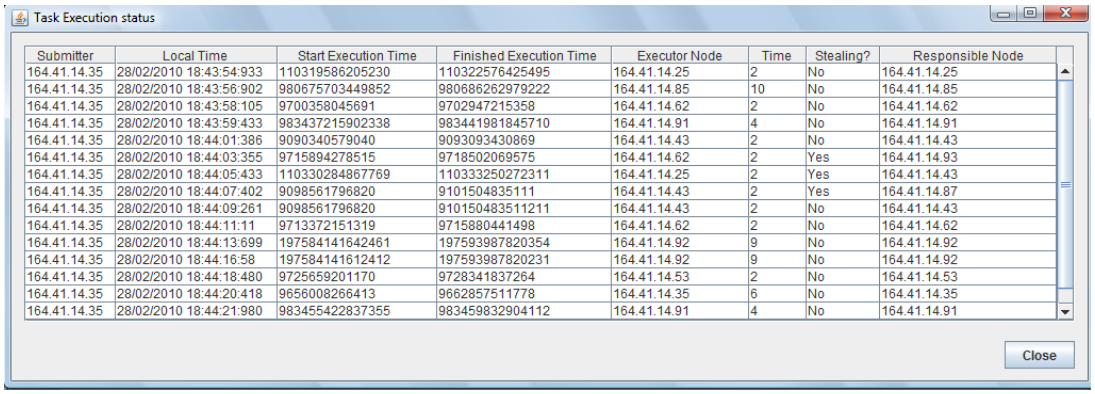


Figura 5.4: Speedup da política de work stealing WSR.



Submitter	Local Time	Start Execution Time	Finished Execution Time	Executor Node	Time	Stealing?	Responsible Node
164.41.14.35	28/02/2010 18:43:54:933	110319586205230	110322576425495	164.41.14.25	2	No	164.41.14.25
164.41.14.35	28/02/2010 18:43:56:902	980675703449852	980686262979222	164.41.14.85	10	No	164.41.14.85
164.41.14.35	28/02/2010 18:43:58:105	9700358045691	9702947215358	164.41.14.62	2	No	164.41.14.62
164.41.14.35	28/02/2010 18:43:59:433	983437215902338	983441981845710	164.41.14.91	4	No	164.41.14.91
164.41.14.35	28/02/2010 18:44:01:386	9090340579040	9093093430869	164.41.14.43	2	No	164.41.14.43
164.41.14.35	28/02/2010 18:44:03:355	9715894278515	9718502069575	164.41.14.62	2	Yes	164.41.14.93
164.41.14.35	28/02/2010 18:44:05:433	110330284867769	110333250272311	164.41.14.25	2	Yes	164.41.14.43
164.41.14.35	28/02/2010 18:44:07:402	9098561796820	9101504835111	164.41.14.43	2	Yes	164.41.14.87
164.41.14.35	28/02/2010 18:44:09:261	9098561796820	91015048351121	164.41.14.43	2	No	164.41.14.43
164.41.14.35	28/02/2010 18:44:11:11	9713372151319	9715880441498	164.41.14.62	2	No	164.41.14.62
164.41.14.35	28/02/2010 18:44:13:699	197584141642461	197593987820354	164.41.14.92	9	No	164.41.14.92
164.41.14.35	28/02/2010 18:44:16:58	197584141612412	197593987820231	164.41.14.92	9	No	164.41.14.92
164.41.14.35	28/02/2010 18:44:18:480	9725659201170	9728341837264	164.41.14.53	2	No	164.41.14.53
164.41.14.35	28/02/2010 18:44:20:418	9656008266413	9662857511778	164.41.14.35	6	No	164.41.14.35
164.41.14.35	28/02/2010 18:44:21:980	983455422837355	983459832904112	164.41.14.91	4	No	164.41.14.91

Figura 5.5: Ferramenta de monitoramento da execução das tarefas.

Capítulo 6

Conclusão e Trabalhos Futuros

A presente dissertação apresentou o projeto e a avaliação de um framework flexível para alocação descentralizada de tarefas em ambiente de *grid*, além de duas variantes da política de *work stealing*: *Work Stealing With Replication* (WSR) e *Local Work Stealing With Replication* (LWSR). O framework proposto foi implementado utilizando o JXTA (JXTA, 2003) como *middleware* P2P e o Chord (Stoica et al., 2001) como *overlay* P2P. Nesse trabalho, desenvolvemos toda a especificação do Chord em Java, e também, uma ferramenta para monitorar a execução das aplicações, a qual possibilita controlar visualmente a execução das aplicações.

Para a coleta e análise dos dados experimentais utilizamos um *desktop grid* heterogêneo composto de 16 máquinas distribuídas em dois laboratórios distintos e desenvolvemos uma aplicação para balanceamento de tráfego aéreo, utilizando dados reais de movimentos aéreos dos dias dias 30 de abril e 02 de maio de 2008 fornecidos pelo Primeiro Centro Integrado de Defesa Aérea e Controle de Tráfego Aéreo (CINDACTA I), composta por 90, 180 ou 270 tarefas, de modo a analisar o comportamento das políticas de *work stealing* quando contrastadas com a execução das tarefas sem o uso de *work stealing*.

Os resultados obtidos mostram que conseguimos um bom ganho de desempenho aos utilizarmos políticas de *work stealing*. Na política LWSR observamos que o tempo total de execução da aplicação é reduzido quando o número de nós é maior que quatro. Na política WSR o tempo de execução foi sempre menor, quando comparado ao tempo gasto ao não utilizarmos uma política de *work stealing*. Utilizamos o tempo de execução da melhor máquina para validarmos o ganho das políticas e com isso, o *speedup* obtido para 180 tarefas com 16 máquinas foi de $11,86$, reduzindo o tempo de execução de $22,35$ minutos para $1,8$ minutos.

Como trabalhos futuros a serem realizados de modo a dar continuidade às atividades desenvolvidas e descritas nessa dissertação, podemos citar:

1. Incorporar novas políticas de alocação de tarefas ao protótipo;
2. Adicionar formas dos nós aprenderem sobre o roubo de tarefas;
3. Incorporar mecanismos de tolerância a falhas ao framework, como por exemplo, *checkpoints*;
4. Investigar a escalabilidade do framework, identificando quais parâmetros contribuem para o custo de um roubo de tarefas;

5. Realizar experimentos em uma rede distribuída de grande porte como *PlanetLab*.

Referências

Abramson, D., Giddy, J., and Kotler, L. (2000). High performance parametric modeling with nimrod/g: Killer application for the global grid? *Parallel and Distributed Processing Symposium, International*, 0:520. 1, 26

Accord (2009). http://accord.dev.java.net. 17

Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. (1993). *Network Flows: theory, algorithms, and applications*. Prentice-Hall, 1th edition. viii, 73, 74, 75, 76, 77

Ahuja, R. K. and Orlin, J. B. (1989). A fast and simple algorithm for the maximum flow problem. *Operations Research*, 37:748–759. xii, 54, 77, 79, 84, 85

Ajmani, S., Clarke, D. E., Moh, C.-H., and Richman, S. (2002). Conchord: Cooperative sdsi certificate storage and name resolution. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 141–154, London, UK. Springer-Verlag. 12

Alon, N. (1990). Generating pseudo-random permutations and maximum flow algorithms. *Inf. Process. Lett.*, 35(4):201–204. 83

Altschul, S. F., Gish, W., Miller, W., Myers, E. W., and Lipman, D. J. (1990). Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410. 28

Aspnes, J. and Shah, G. (2003). Skip graphs. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 384–393, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics. 15, 20, 36, 38, 40

Aspnes, J. and Shah, G. (2007). Skip graphs. *ACM Trans. Algorithms*, 3(4):37. vii, 15, 16

Assis, L., Nóbrega-Júnior, N., Brasileiro, F., and Cirne, W. (2006). Uma heurística de particionamento de carga divisível para grids computacionais. *XXIV Simpósio Brasileiro de Redes de Computadores*. 25

Awan, A., Ferreira, R. A., Jagannathan, S., and Grama, A. (2006). Unstructured peer-to-peer networks for sharing processor cycles. *Parallel Comput.*, 32(2):115–135. xi, 2, 33, 34, 37, 39

Balls, G. T., Baden, S. B., Kispersky, T., Bartol, T. M., and Sejnowski, T. J. (2004). A large scale monte carlo simulator for cellular microphysiology. *Parallel and Distributed Processing Symposium, International*, 1:42a. 26

Barcellos, M. P. and Gaspary, L. P. (2007). Fundamentos, tecnologias e tendências rumo a redes p2p seguras. In Breitman, K. and Anido, R., editors, *Atualizações em Informática*, pages 187–243. Editora PUC RIO. 9

Berman, F., Wolski, R., Casanova, H., Cirne, W., Dail, H., Faerman, M., Figueira, S., Hayes, J., Obertelli, G., Schopf, J., Shao, G., Smallen, S., Spring, N., Su, A., and Zagorodnov, D. (2003). Adaptive computing on the grid using apples. *IEEE Trans. Parallel Distrib. Syst.*, 14(4):369–382. 1

Bertsimas, D. and Patterson, S. S. (2000). The traffic flow management rerouting problem in air traffic control: A dynamic network flow approach. *Transportation Science*, 34(3):239–255. 3

Blumofe, R. D. and Leiserson, C. E. (1999). Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748. 2, 3, 29

Blumofe, R. D. and Lisiecki, P. A. (1997). Adaptive and reliable parallel computing on networks of workstations. In *ATEC '97: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 10–10, Berkeley, CA, USA. USENIX Association. 29

Bölöni, Turgut, D., and Marinescu, D. C. (2005). n-Cycle: a set of algorithms for task distribution on a commodity grid. In *IEEE International Symposium on Cluster Computing and the Grid CCGrid 2005*. 37

Bölöni, L., Turgut, D., and Marinescu, D. C. (2006). Task distribution with a random overlay network. *Future Gener. Comput. Syst.*, 22(6):676–687. vii, xi, 34, 35, 38, 39

Campos, V. B. G. (2009). Algoritmos para Resolução de Problemas em Redes. *Notas de aula, disponível on-line em: http://www.ime.eb.br/~webde2/prof/vania/apostilas/Apostila-Redes.pdf*. 80

Casanova, H., Zagorodnov, D., Berman, F., and Legrand, A. (2000). Heuristics for scheduling parameter sweep applications in grid environments. In *HCW '00: Proceedings of the 9th Heterogeneous Computing Workshop*, page 349, Washington, DC, USA. IEEE Computer Society. 26

Casavant, T. and Kuhl, J. (1988). A taxonomy of scheduling in general-purpose distributed computing systems. *Software Engineering, IEEE Transactions on*, 14(2):141–154. vii, 23, 24, 25

Chakravarti, A., Baumgartner, G., and Lauria, M. (2005). The organic grid: self-organizing computation on a peer-to-peer network. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 35(3):373–384. xi, 32, 37, 38, 39

Chakravarti, A. J., Baumgartner, G., and Lauria, M. (2004). The organic grid: Self-organizing computation on a peer-to-peer network. *Autonomic Computing, International Conference on*, 0:96–103. 1

Chang, H. W. D. and Oldham, W. J. B. (1995). Dynamic task allocation models for large distributed computing systems. *IEEE Trans. Parallel Distrib. Syst.*, 6(12):1301–1315. 23, 25

Chen, Y., Katz, R. H., and Kubiatowicz, J. (2002). Scan: A dynamic, scalable, and efficient content distribution network. In *Pervasive '02: Proceedings of the First International Conference on Pervasive Computing*, pages 282–296, London, UK. Springer-Verlag. 15

Cheriyan, J. and Hagerup, T. (1989). A randomized maximum-flow algorithm. *Symposium on Foundations of Computer Science*, 0:118–123. 83

Cheriyan, J., Hagerup, T., and Mehlhorn, K. (1990). Can a maximum flow be computed on o(nm) time? In *ICALP '90: Proceedings of the 17th International Colloquium on Automata, Languages and Programming*, pages 235–248, London, UK. Springer-Verlag. 83

Cirne, W., Brasileiro, F., Andrade, N., Costa, L. B., Alisson Andrade and, R. N., , and Mowbray, M. (2006). Labs of the world, unite!!! *Journal of Grid Computing*, 4(3):225–246. vii, x, 17, 18, 21, 22

Cirne, W., Paranhos, D., Costa, L., Santos-Neto, E., Brasileiro, F., Sauvé, J., Silva, F. A. B., Barros, C. O., and Silveira, C. (2003). Running bag-of-tasks applications on computational grids: The mygrid approach. *Parallel Processing, International Conference on*, 0:407. 25, 26

Clarke, I., Sandberg, O., Wiley, B., and Hong, T. W. (2001). Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009:46–66. 7, 9

Cohen, B. (2003). Incentives build robustness in bittorrent. *In Proceedings of the 1st Workshop on Economics of Peer-to-Peer Systems*. 7

Cooper, K., Dasgupta, A., Kennedy, K., Koelbel, C., Marin, G., Mazina, M., Mellor-crummey, J., Berman, F., Casanova, H., Chien, A., Dail, H., Liu, X., Olugbile, A., Sievert, O., Xia, H., Johnsson, L., Liu, B., Patel, M., Reed, D., Deng, W., and Mendes, C. (2004). New grid scheduling and rescheduling methods in the grads project. In *in Proceedings of NSF Next Generation Software Workshop:International Parallel and Distributed Processing Symposium. Santa Fe, USA: IEEE CS*, pages 209–229. Press. 26, 27

Cormen, T. H., Leiserson, C. E., Rivest, R. E., and Stein, C. (1998). *Introduction to Algorithms*. MIT Press, 2th edition. 73, 74, 76, 79, 80

Couloris, G., Dollimore, J., and Kindberg, T. (2007). *Sistemas Distribuídos: conceitos e práticas*. Bookman, 4 edition. 7

Cox, R., Muthitacharoen, A., and Morris, R. T. (2002). Serving dns using a peer-to-peer lookup service. *In First International Workshop on Peer-to-Peer Systems(Cambridge)*. 12

Dabek, F., Kaashoek, M. F., Karger, D., Morris, R., and Stoica, I. (2001). Wide-area cooperative storage with cfs. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 202–215, New York, NY, USA. ACM. 12

Dabek, F., Zhao, B., Druschel, P., Kubiatowicz, J., and Stoica, I. (2003). Towards a common api for structured peer-to-peer overlays. In *The 2nd International Workshop on Peer-to- Peer Systems (IPTPS'03)*, pages 33–44. 17

Dantzig, G. B. and Fulkerson, D. R. (1956). On the max-flow min-cut theorem of networks. In Kuhn, H. W. and Tucker, A. W., editors, *Linear Inequalities and Related Systems*, number 2 in Annals of Mathematics Study 38, pages 215–221. Princeton University Press, 1 edition. 77

Dinan, J., Larkins, D. B., Sadayappan, P., Krishnamoorthy, S., and Nieplocha, J. (2009). Scalable work stealing. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA. ACM. 2

Dingledine, R., Freedman, M. J., and Molnar, D. (2000). The free haven project: Distributed anonymous storage service. In *In Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, pages 67–95. 8

Dinic, E. A. (1970). Algorithm for solution of a problem of maximum flow in a network with power estimation. *Sovietic Mathematics Docklady*, 11(5):1277–1280. 54, 78, 80

Druschel, P. and Rowstron, A. (2001). Past: A large-scale, persistent peer-to-peer storage utility. In *In HotOS VIII*, pages 75–80. 9, 10

Edmonds, J. and Karp, R. M. (1972). Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264. 54, 78, 80, 84

Evans, J. R. and Minieka, E. (1992). *Optimization Algorithms for Networks and Graphs*. Marcel Dekker. viii, 74, 79

Ford, L. and Fulkerson, D. (1957). A simple algorithm for finding maximal network flows and an application to the hitchcock problem. *Can. J. Math*, 9:210–218. 78

Ford, L. R. and Fulkerson, D. R. (1956). Maximal flow through a network. *Naval Research Logistics Quarterly*, 8:399–404. 77

Ford, L. R. J. and Fulkerson, D. R. (1962). *Flows in Networks*. Priceton University Press. 73, 79, 80

Forestiero, A. and Mastroianni, C. (2009). A swarm algorithm for a self-structured p2p information system. *Trans. Evol. Comp*, 13(4):681–694. 2

Foster, I. and Iamnitchi, A. (2003). On death, taxes and the convergence of peer-to-peer and grid computing. In *In 2nd Internation Workshop on Peer-to-Peer Systems (IPTPS'03)*, pages 118–128. 2, 16, 17

Foster, I., Kesselman, C., and Tuecke, S. (2001). The anatomy of the grid: Enabling scalable virtual organizations. *Int. J. High Perform. Comput. Appl.*, 15(3):200–222. 1

Fujiwara, K., Teranishi, Y., Takeuchi, S., Harumoto, K., and Nishio, S. (2008). An implementation of lightweight message transport mechanism for p2p agent platform on ad-hoc networks. In *SAINT '08: Proceedings of the 2008 International Symposium on Applications and the Internet*, pages 361–364, Washington, DC, USA. IEEE Computer Society. vii, x, 17, 20, 21, 22

Fulkerson, D. and Dantzig, G. (1955). Computations of maximum flow in networks. *Naval Research Logistics Quarterly*, 2(2):277–283. 77

Gabow, H. N. (1985). Scaling algorithms for network problems. *J. Comput. Syst. Sci.*, 31(2):148–168. 78, 84

Genome@Home (2009). The genome@home. http://genomeathome.stanford.edu. 6

Gkantsidis, C., Mihail, M., and Saberi, A. (2006). Random walks in peer-to-peer networks: algorithms and evaluation. *Perform. Eval.*, 63(3):241–263. 8, 29

Gnutella (2000). The gnutella. http://www.gnutella.wego.com. 5, 7, 17

Goldberg, A. V. and Tarjan, R. E. (1988). A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940. 78, 79, 80, 81, 84

Graham, L. R., Lawler, E. L., Lenstra, J. K., and Kan, A. H. G. R. (1979). Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Math*, 5:287–326. 2, 23, 25

Grimshaw, A. S., Wulf, W. A., and The Legion Team, C. (1997). The legion vision of a worldwide virtual computer. *Commun. ACM*, 40(1):39–45. 1

Groove (2000). Groove peer computing platform, groove networks inc. http://www.groove.net. 17

Hand, S. and Roscoe, T. (2002). Mnemosyne: Peer-to-peer steganographic storage. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 130–140, London, UK. Springer-Verlag. 15

Hummel, S. F., Schmidt, J., Uma, R. N., and Wein, J. (1996). Load-sharing in heterogeneous systems via weighted factoring. In *SPAA '96: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pages 318–328, New York, NY, USA. ACM. 25

Hummel, S. F., Schonberg, E., and Flynn, L. E. (1992). Factoring: a method for scheduling parallel loops. *Commun. ACM*, 35(8):90–101. 25

Jaffar, J., Santosa, A. E., Yap, R. H. C., and Zhu, K. Q. (2004). Scalable distributed depth-first search with greedy work stealing. In *ICTAI '04: Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence*, pages 98–103, Washington, DC, USA. IEEE Computer Society. 2

Jain, R. K., Chiu, D.-M. W., and Hawe, W. R. (1984). A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. Technical report, Digital Equipment Corporation. 31

Johnson, R. E. (1997). Frameworks = (components + patterns). *Commun. ACM*, 40(10):39–42. 17

JTella (2009). The jtella. http://jtella.sourceforge.net. 17

JXTA (2001-2003). v2.0 protocols specification sun microsystems, the internet society. http://www.jxta.org. 3, 17, 19, 21, 40, 42, 61

Kaneko, Y., Harumoto, K., Fukumura, S., Shimojo, S., and Nishio, S. (2005). A location-based peer-to-peer network for context-aware services in a ubiquitous environment. In *SAINT-W '05: Proceedings of the 2005 Symposium on Applications and the Internet Workshops*, pages 208–211, Washington, DC, USA. IEEE Computer Society. 20

Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., and Lewin, D. (1997). Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663, New York, NY, USA. ACM. 10

Karzanov, A. V. (1974). Determining the maximal flow in a network by the method of preflows. *Soviet Math*, 15:434–437. 78, 80

King, V., Rao, S., and Tarjan, R. (1992). A faster deterministic maximum flow algorithm. In *SODA '92: Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms*, pages 157–164, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics. xii, 79, 83, 84

Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., and Zhao, B. (2000). Oceanstore: an architecture for global-scale persistent storage. *SIGPLAN Not.*, 35(11):190–201. 10, 15

Kumar, V., Grama, A. Y., and Vempaty, N. R. (1994). Scalable load balancing techniques for parallel computers. *J. Parallel Distrib. Comput.*, 22(1):60–79. 29

Leibowitz, N., Ripeanu, M., and Wierzbicki, A. (2003). Deconstructing the kazaa network. In *WIAPP '03: Proceedings of the The Third IEEE Workshop on Internet Applications*, page 112, Washington, DC, USA. IEEE Computer Society. 7

Lo, V. M. (1988). Heuristic algorithms for task assignment in distributed systems. *IEEE Trans. Comput.*, 37(11):1384–1397. 24, 25

Maymounkov, P. and Mazieres, D. (2002). Kademlia: A peer-to-peer information system based on the xor metric. 7, 10

Mendes, H., Weigang, L., Boukerche, A., and Melo, A. C. M. A. (2009). Bag-of-tasks self-scheduling over range-queriable search overlays. In *6th IFIP International Conference on Network and Parallel Computing*, volume 6, pages 109–116. vii, xi, 2, 16, 25, 36, 37, 39, 44

Michael, M. M., Vechev, M. T., and Saraswat, V. A. (2009). Idempotent work stealing. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 45–54, New York, NY, USA. ACM. 29

Nejdl, W., Wolf, B., Qu, C., Decker, S., Sintek, M., Naeve, A., Nilsson, M., Palmér, M., and Risch, T. (2002). Edutella: a p2p networking infrastructure based on rdf. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*, pages 604–615, New York, NY, USA. ACM. 8

Netto, P. O. B. (2003). *Grafos: Teoria Modelos, Algoritmos*. Edgard Blücher, 3th edition. 74, 80

Open Overlay Project (2004). E-science project. http://www.comp.lancs.ac.uk/computing/research/mpg/projects/openoverlays/. 17

OpenNap (2001). Open source napster server. http://opennap.sourceforge.net. 7, 8

Park, G.-L., Shirazi, B., Marquis, J., and Choo, H. (1997). Decisive path scheduling: A new list scheduling method. In *ICPP '97: Proceedings of the international Conference on Parallel Processing*, pages 472–480, Washington, DC, USA. IEEE Computer Society. 25

Plaxton, C. G., Rajaraman, R., and Richa, A. W. (1997). Accessing nearby copies of replicated objects in a distributed environment. In *SPAA '97: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 311–320, New York, NY, USA. ACM. 13, 15

Polychronopoulos, C. D. and Kuck, D. J. (1987). Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Trans. Comput.*, 36(12):1425–1439. 25, 28

Portmann, M., Ardon, S., Senac, P., and Seneviratne, A. (2004). Prost: A programmable structured peer-to-peer overlay network. *Peer-to-Peer Computing, IEEE International Conference on*, 0:280–281. 17

Pugh, W. (1990). Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676. 15

Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Schenker, S. (2001). 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, New York, NY, USA. ACM. 7, 9, 40

Repantis, T., Drougas, Y., and Kalogeraki, V. (2005). Adaptive resource management in peer-to-peer middleware. *Parallel and Distributed Processing Symposium, International*, 3:132b. vii, x, 30, 31, 32, 37, 39

Rowstron, A. and Druschel, P. (2001). Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350. 7, 10

Ruhe, G. (1991). *Algorithmic Aspect of Flows in Networks*. Kluwer Academic Publishers. 77, 84

Seti@Home (2009). The seti@home. http://setiathome.berkeley.edu. 6

Shao, G. (2001). *Adaptive Scheduling of Master/Worker Applications on Distributed Computational Resources*. (PhD) Dissertation, Univ. California at San Diego. 25, 27, 38

Shen, J. and Yuan, S. (2008). Adaptive task allocation for p2p-based e-services composition. volume 1, pages 313–318. xi, 25, 36, 39

Shiloach, Y. and Vishkin, U. (1982). An o(n^2 log n) parallel max-flow algorithm. *J. Algorithms*, 3(2):128–146. 78

Shirky, C. (2000). What is p2p... and what isn't. http://www.oreillynet.com/pub/a/p2p/2000/11/24/shirky1-whatisp2p.html. 5

Sleator, D. D. and Tarjan, R. E. (1981). A data structure for dynamic trees. In *STOC '81: Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 114–122, New York, NY, USA. ACM. 82

Smallen, S., Crine, W., Frey, J., Berman, F., Wolski, R., Su, M.-H., Kesselman, C., Young, S., and Ellisman, M. (2000). Combining workstations and supercomputers to support grid applications: the parallel tomography experience. In *Heterogeneous Computing Workshop, 2000. (HCW 2000) Proceedings. 9th*, pages 241–252. 26

Sousa, M. S. and Melo, A. C. M. A. (2006). Packageblast: an adaptive multi-policy grid service for biological sequence comparison. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 156–160, New York, NY, USA. ACM. 28, 37, 38, 39

Stephanos and Spinellis, D. (2004). A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv.*, 36(4):335–371. vii, ix, 2, 5, 6, 7, 8, 9, 10, 14

Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. (2001). Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA. ACM. 3, 7, 9, 10, 11, 12, 17, 40, 45, 61

Stoica, I., Morris, R., Liben-Nowell, D., Karger, D. R., Kaashoek, M. F., Dabek, F., and Balakrishnan, H. (2003). Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32. vii, 12, 13

Tang, P. and Yew, P.-C. (1986). Processor self-scheduling for multiple-nested parallel loops. *International Conference on Parallel Processing (ICPP'86)*, pages 528–535. 25, 27

Tarjan, R. E. (1983). *Data Structures and Network Algorithms*. CBMS-NSF Regional Conference Series in Applied Mathematics. Socienty for Industrial and Applied Mathematics. 82

Tarjan, R. E. (1989). Algorithms for maximum network flow. *Mathematical Methods of Operations Research*, 33:21–37. 77

Taylor, I., Shields, M., and Wang, I. (2004). Resource management of the triana peer-to-peer services. pages 451–462. 1

Treleaven, K. and Mao, Z.-H. (2008). Conflict resolution and traffic complexity of multiple intersecting flows of. *IEEE Transactions on Intelligent Transportation Systems*, 9(4):633 – 643. 3

Tsoumakos, D. and Roussopoulos, N. (2003). A comparison of peer-to-peer search methods. In *6th International Workhop on the Web and Databases*. 8

Tzen, T. H. and Ni, L. M. (1993). Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers. *IEEE Trans. Parallel Distrib. Syst.*, 4(1):87–98. 25

van Nieuwpoort, R. V., Kielmann, T., and Bal, H. E. (2001). Efficient load balancing for wide-area divide-and-conquer applications. In *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 34–43, New York, NY, USA. ACM. 29

Waldman, M., Rubin, A. D., and Cranor, L. F. (2000). Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *In Proc. 9th USENIX Security Symposium*, pages 59–72. 8

Walkerdine, J., Hughes, D., Rayson, P., Simms, J., Gilleade, K., Mariani, J., and Sommerville, I. (2008). A framework for p2p application development. *Comput. Commun.*, 31(2):387–401. vii, x, 2, 16, 17, 19, 20, 21, 22

Waslander, S. L., Raffard, R. L., and Tomlin, C. J. (2006). Toward efficient and equitable air traffic flow control. In *Proceedings of the AACC American Control Conference*, pages 5189–5194. IEEE Computer Society. 3

Wirfs-Brock, R. J. and Johnson, R. E. (1990). Surveying current research in object-oriented design. *Commun. ACM*, 33(9):104–124. 17

Yau, S. and Satish, V. (1993). A task allocation algorithm for distributed computing systems. pages 336–342. 25

Zhao, B. Y., Huang, L., Stribling, J., Rhea, S. C., Joseph, A. D., and Kubiatowicz, J. D. (2004). Tapestry: a resilient global-scale overlay for service deployment. *Selected Areas in Communications, IEEE Journal on*, 22(1):41–53. 7, 9, 10, 13, 14

Anexo I

Algoritmos de Fluxo de Rede

I.1 Definição

Um fluxo de rede pode ser definido como um grafo orientado $G(V, E)$ onde V é um conjunto v de vértices e E um conjunto u de arestas orientadas com dois vértices especiais, uma origem s e um destino t, também conhecido como sorvedouro. Cada aresta $(u, v) \in E$ tem um custo que denota o custo por unidade de fluxo da aresta, que varia linearmente com o total do fluxo e uma capacidade c não negativa ($c(u, v) > 0$) em toda aresta (u, v) que representa o fluxo máximo admissível da aresta. Se $(u, v) \notin E$, então $c(u, v) = 0$ (Ahuja et al., 1993; Cormen et al., 1998; Ford and Fulkerson, 1962).

Um fluxo em G é uma função de valor real $f : V\ x\ V \rightarrow \Re$ que satisfaz a três propriedades (Ahuja et al., 1993):

1. **Restrição de capacidade da aresta**: $f(u, v) \leq c(u, v)\ \forall u, v \in V$;
2. **Anti-simetria oblíqua**: $f(u, v) = -f(v, u)\ \forall u, v \in V$;
3. **Conservação de fluxo**: $\sum_{v \in V} f(u, v) = 0\ \forall u \in V - \{s, t\}$ e $v \in V - \{s, t\}$.

A quantidade de fluxo $f(u, v)$, que pode ser positiva, zero ou negativa é chamada de fluxo do vértice u até o vértice v é definida como:

$$|f| = \sum_{v \in V} f(s, v) \tag{I.1}$$

Por anti-simetria, a propriedade de conservação do fluxo pode ser reescrita como:

$$\sum_{u \in V} f(u, v) = 0\ \forall\ v \in V - \{s, t\} \tag{I.2}$$

Quando nem (u, v) nem (v, u) estão em E, então não há nenhum fluxo entre u e v e $f(u, v) = f(v, u) = 0$.

Além disso, cabe ressaltar que as propriedades do fluxo lidam com fluxos que são positivos. O fluxo total positivo que entra em um vértice v é definido como:

$$\sum_{u \in V_{f(u,v)>0}} f(u,v) \tag{I.3}$$

A figura I.1 ilustra uma rede de fluxo com origem s e destino t e quatro vértices intermediários, que satisfaz as propriedades de conservação do fluxo e as restrições da capacidade. Como pode ser observado, a capacidade de nenhuma aresta é excedida e a conservação do fluxo mantém-se a cada nó, de modo que, o fluxo que entra em um vértice, sai dele na mesma taxa que entrou (Cormen et al., 1998; Netto, 2003). Nessa rede, o fluxo mínimo de s para t é de duas unidades, $s \rightarrow 1 \rightarrow 2 \rightarrow t$.

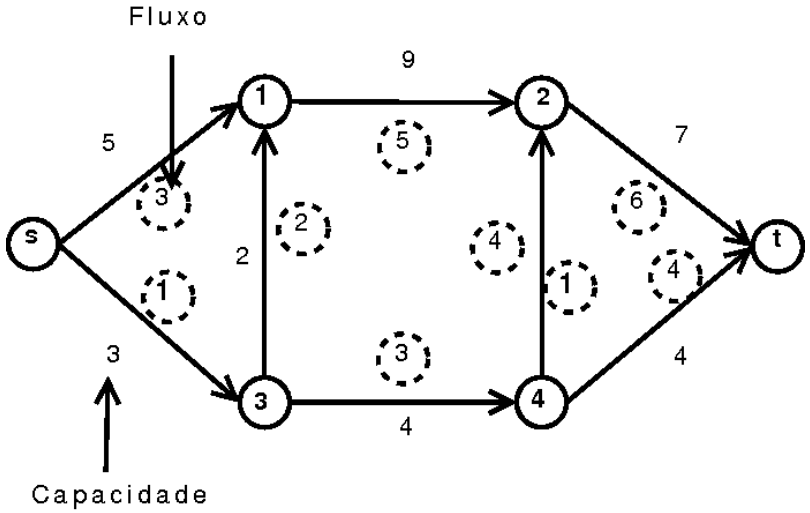


Figura I.1: Exemplo de rede de fluxo. Baseada em (Evans and Minieka, 1992)

I.2 Representação de Fluxo de Rede

Segundo (Ahuja et al., 1993) o desempenho de um algoritmo de rede depende não somente do algoritmo, mas também da forma utilizada para representar a rede no computador e do esquema utilizado para manter e atualizar os resultados intermediários. A representação da rede associada à estrutura dos dados pode contribuir para melhorar o tempo de execução de um algoritmo.

I.2.1 Matriz de incidência

A representação de matriz de incidência representa uma rede como uma matriz de restrição de problemas de custo mínimo. Essa representação armazena a rede como uma matriz $n\ x\ m$ tendo uma linha para cada vértice da rede e uma coluna para cada aresta. A coluna correspondente à aresta (u, v) tem somente elementos

não nulos: $a+1$ na linha correspondente ao vértice u e $a-1$ na linha correspondente ao vértice v. A tabela I.1 é a representação da rede apresentada na figura I.2.

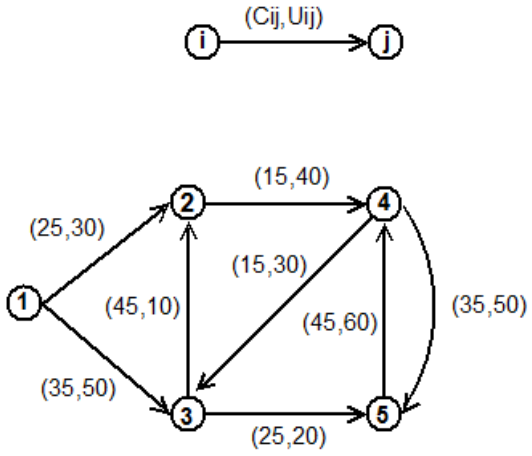


Figura I.2: Exemplo de rede. Baseada em (Ahuja et al., 1993)

	(1,2)	(1,3)	(2,4)	(3,2)	(4,3)	(4,5)	(5,3)	(5,4)
1	1	1	0	0	0	0	0	0
2	-1	0	1	-1	0	0	o	0
3	0	-1	0	1	-1	0	-1	0
4	0	0	-1	0	1	1	0	-1
5	0	0	0	0	0	-1	1	1

Tabela I.1: Matriz de incidência da rede de exemplo da figura I.2.

Para (Ahuja et al., 1993) essa representação da rede não é eficiente do ponto de vista do espaço, pois, a matriz possui poucos coeficientes não zeros.

I.2.2 Matriz de Adjacência

A representação de matriz de adjacência armazena a rede como uma matriz $n\ x\ n$ $M = \{h_{ij}\}$. A matriz possui uma linha e uma coluna correspondente à cada vértice e sua ij*–ésima* entrada h_{ij} é igual a 1 se $(i, j) \in E$ ou igual a zero, caso contrário. A tabela I.2 é a representação da rede apresentada na figura I.2.

	1	2	3	4	5
1	0	1	1	0	0
2	0	0	0	1	0
3	0	1	0	0	0
4	0	0	1	0	1
5	0	0	1	1	0

Tabela I.2: Matriz de adjacência da rede de exemplo da figura I.2.

A matriz de adjacência possui n^2 elementos dos quais apenas m são zero. Conseqüentemente, essa representação é eficiente do ponto de vista do espaço, apenas se a rede for suficientemente densa. Entretanto, a simplicidade dessa representação permite usá-la facilmente na implementação, da maioria dos algoritmos de rede. Nessa representação, para determinar o custo ou a capacidade de qualquer aresta (i, j) é necessário simplesmente consultar o $ij-$*ésimo* elemento da matriz.

I.2.3 Lista de adjacência

A representação de lista de adjacência de uma rede consiste em um arranjo adjacente de $|V|$ listas ligadas, uma para cada cada vértice em V. Para $u \in V$, a lista de adjacências contém referência para todos os vértices v tal que existe uma aresta $(u, v) \in E$. Isto é, $adj[u]$ consiste em todos os vértices adjacentes a *u* em G.

A figura I.3 é a representação da lista de adjacência da figura I.2.

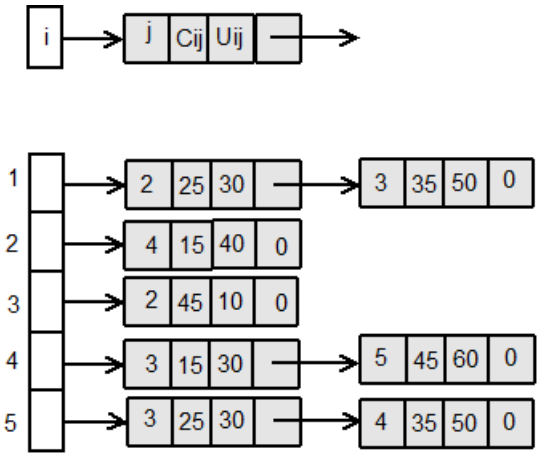


Figura I.3: Lista de adjacência da rede de exemplo da figura I.2.

Para (Cormen et al., 1998), uma desvantagem da representação da lista de adjacência é que não existe nenhum modo rápido de determinar se uma aresta (u, v) está presente no grafo, sem procurar v na lista de adjacências.

I.3 Tipos de problemas

As redes fazem parte do cotidiano da vida humana. Elas estão presentes por toda parte, como por exemplo, rede de eletricidade, rede de telefone, rede de distribuição da produção, rede de computadores. Em todos esses domínios, o objetivo é mover alguma entidade (eletricidade, um produto, veículo ou uma pessoa) de um ponto para o outro em uma rede subjacente, de modo eficiente tanto do ponto de vista da prestação de serviço ao usuário quando da utilização da rede (Ahuja et al., 1993).

Rede de fluxo é um problema de domínio que se coloca entre várias áreas incluindo a matemática, ciência da computação, engenharia, gerenciamento de recursos e pesquisa operacional, que utilizam do conceito de rede para obter repostas aos seguintes questionamentos:

1. **Problema do caminho mais curto**. Qual o melhor caminho para atravessar uma rede saindo de um ponto (origem) e chegando em outro (destino) com o menor custo possível?
2. **Problema do fluxo máximo**. Em uma rede com arestas limitada pela capacidade do fluxo, como obter o maior fluxo admissível entre dois pontos da rede sem violar a sua capacidade?
3. **Problema do custo mínimo**. Admita que foi atribuído um custo por unidade de fluxo em uma rede com capacidade nas arestas e que deseja-se enviar alguma unidade de material que encontra-se em um ou mais pontos da rede para algum outro ponto da rede. Como esse material pode ser enviado com o menor custo possível?

Segundo (Ahuja et al., 1993) a solução puramente matemática desses problemas é trivial. Basta enumerar o conjunto de soluções possíveis e escolher aquela que é a melhor. Entretanto, a enumeração das alternativas possíveis pode ser proibitiva do ponto de vista prático, de modo que é melhor utilizar de algoritmos que na média apresentam bons resultados. Para (Ruhe, 1991) o desenvolvimento de algoritmos eficientes para muitas das classes de problemas de rede combinado com a eficácia dos computadores tem contribuído para o aumento de aplicações que fazem uso de fluxo de rede.

Um exemplo de fluxo em rede é uma rede de dutos ao longo dos quais pode-se bombear óleo. O objetivo é bombear o máximo possível de óleo de uma fonte até um destino sem exceder a capacidade de nenhum dos dutos. Nessa dissertação, vamos nos concentrar na resolução do problema de fluxo máximo.

I.3.1 Problema do fluxo máximo

Para (Ahuja and Orlin, 1989) o problema do fluxo máximo é um dos problemas fundamentais da teoria de fluxo de rede e por isso tem sido extensivamente investigado. Para (Tarjan, 1989), o problema de fluxo de rede é um dos problemas clássicos de otimização de rede. Do ponto de vista da complexidade, problema de fluxo máximo é também um dos mais intrigantes, devido à variedade e riqueza de estruturas dos algoritmos propostos para resolvê-lo. O problema do caminho mais curto modela as situações em que o custo incorre sobre o fluxo, mas sem estar sujeito a restrições da capacidade das arestas. Em contrapartida, no problema do fluxo máximo não incorre nenhum custo, mas está restrito a capacidade das arestas. O problema de fluxo máximo então, procura por uma solução viável que possibilita "fluir" a maior quantidade de fluxo de um vértice s a outro vértice t. Exemplos de problemas de fluxo incluem determinar o máximo fluxo (1) de mensagens em uma rede de telecomunicação, (2) de corrente em uma rede elétrica, (3) de carros em uma rede de estradas.

O problema de fluxo máximo foi primeiro formulado por (Fulkerson and Dantzig, 1955) e (Dantzig and Fulkerson, 1956) e solucionado por (Ford and Fulkerson, 1956) utilizando o conceito de fluxo em caminhos de incremento. Desde então, uma variedade de algoritmos tem sido desenvolvidos para este problema, muito

dos quais apresentados na tabela I.3. Nessa tabela, n é o número de vértices, m é o número de arestas.

Número	Algoritmo	Tempo de Execução
1	Ford and Fulkerson (1956)	$O(nmU)$
2	Edmonds and Karp (1972)	$O(nm^2)$
3	Dinic (1970)	$O(n^2m)$
4	Karzanov (1974)	$O(n^3)$
5	Cherkasky (1977)	$O(n^2m^{\frac{1}{2}})$
6	Malhotra and Kumar and Maheshwari (1978)	$O(n^3)$
7	Shiloach (1978)	$O(nm(logn)^2)$
8	Galil (1980)	$O(n^{5/3}m^{2/3})$
9	Galil and Naamad (1980)	$O(nm(logn)^2)$
10	Shiloach and Vishkin (1982)	$O(n^3)$
11	Sleator and Tarjan (1983)	$O(nmlogn)$
12	Tarjan (1984)	$O(n^3)$
13	Gabow (1985)	$O(nmlogU)$
14	Goldberg (1985)	$O(n^3)$
15	Goldeberg and Tarjan (1986)	$O(nmlog(n^2/m))$
16	Ahuja and Orlin (1986)	$O(nm + n^2logU)$
17	Ahuja and Orlin (1988)	$O(nm + (n^2logU)/loglogU)$
18	Ahuja and Orlin and Tarjan (1988)	$O(nm + n^2(logU)^{1/2})$

Tabela I.3: Algoritmos de fluxo máximo

O primeiro algoritmo de fluxo máximo foi desenvolvido por (Ford and Fulkerson, 1957) e baseia-se na procura dos caminhos de incremento, posteriormente modificado por (Edmonds and Karp, 1972) que observaram que o caminho de incremento poderia ser implementado como o caminho mais curto de *s* até *t* em tempo polinomial. Para aumentar a eficiência do algoritmo, (Dinic, 1970) propôs um método para procurar todos os caminhos de incremento em apenas uma fase. Muitos dos algoritmos de fluxo máximo utilizam do método de Dinic. Segundo (Goldberg and Tarjan, 1988), não há uma clara distinção a respeito dos algoritmos apresentados na tabela I.3 que sejam baseados no método de Dinic.

Para (Goldberg and Tarjan, 1988), os algoritmos 3, 6, 10 e 12 são projetados para serem rápidos em grafos densos e os algoritmos 5, 8, 9, 11 e 13 para grafos esparsos. Para grafos densos,o algoritmo com melhor tempo assintótico é da ordem de $O(n^3)$ proposto por (Karzanov, 1974).

Para grafos densos o melhor algoritmo é da ordem de $O(nmlogn)$. Para grafos de pequena densidade, com vértices entre $\Omega(n^2/logn)^3$ e $O(n^2)$, o melhor algoritmo é da ordem de $O(n^{5/3}m^{2/3})$. Para grafos de arestas com capacidade inteira e tamanho moderado, o algoritmo de (Gabow, 1985) possui menor complexidade de tempo. Dentre os algoritmos baseados no método de Dinic, o único algoritmo paralelo é o de (Shiloach and Vishkin, 1982) com tempo de execução da ordem de $O(n^2logn)$ com *n* processos e espaço na ordem de $O(nm)$.

Na presente dissertação, são detalhados os algoritmos de (Goldberg and Tarjan, 1988), (King et al., 1992) e (Ahuja and Orlin, 1989).

I.4 Algoritmos de Fluxo de Rede

I.4.1 Algoritmo genérico de fluxo máximo

A primeira versão do algoritmo de fluxo máximo é o Método de Ford-Fulkerson (Ford and Fulkerson, 1962) que baseia-se na procura dos caminhos de incremento. O método tem por objetivo encontrar um fluxo máximo para uma rede de fluxos. Ele é chamado de método e não algoritmo, devido ao fato de considerar diversos passos com tempos de execução diferentes. O método depende de três considerações importantes para muitos dos problemas associados ao fluxo: as redes residuais, os caminhos de incrementos e os cortes (Cormen et al., 1998).

Em uma rede, as arestas podem ser colocadas em duas categorias (Evans and Minieka, 1992):

- I, o conjunto de arestas cujo fluxo pode ser aumentado;
- R, conjunto de arestas cujo fluxo pode ser diminuído.

As redes residuais (Cormen et al., 1998) consistem em arestas que admitem o aumento do fluxo. Formalmente, considere uma rede de fluxo $G = (V, E)$, com origem *s* e destino *t*. Seja f um fluxo em G e u, v um par de vértices com $u, v \in V$. A quantidade de fluxo adicional que pode-se "empurrar" de u até v e que não ultrapasse a capacidade $c(u, v)$ é a capacidade residual de (u, v) dada por: $c_f(u, v) = c(u, v) - f(u, v)$, ou seja, qualquer aresta que pode ser incluída no conjunto I (Cormen et al., 1998; Evans and Minieka, 1992).

Quando o fluxo $f(u, v)$ é negativo, a capacidade residual $c_f(u, v)$ é maior que a capacidade $c(u, v)$.

Dado uma rede de fluxo $G = (V, E)$ e um fluxo f, a rede residual de G induzida de f é $G_f = (V, E_f)$, onde:

$$E_f = (u, v) \in V \; x \; V : c_f(u, v) > 0 \tag{I.4}$$

Os caminhos de incremento podem ser definidos como um caminho da origem ao destino em que se pode aumentar o fluxo. Esse procedimento pode ser repetido até que não haja mais caminhos de incremento na rede (Cormen et al., 1998).

O método é iterativo e inicia com $f(u, v) = 0 \; \forall u, v \in V$. A cada iteração o fluxo é aumentado encontrando-se um caminho de incremento.

O Algoritmo 5 apresenta o método genérico de Ford-Fulkerson (Ford and Fulkerson, 1962).

Algoritmo 5: Ford-Fulkerson-Method (G,s,t)

```
1 inicializar fluxo f a 0
2 enquanto existir caminho de incremento p faça
3     aumentar fluxo f utilizando p
4 fim enquanto
5 retorne f
```

Esse método é fortemente apoiado pelo teorema do Corte Mínimo/Fluxo Máximo (Cormen et al., 1998; Netto, 2003) que diz o seguinte: se f é um fluxo em uma rede de fluxo G com origem s e destino t, então, as seguintes condições são equivalentes:

1. f é um fluxo máximo em G;
2. A rede residual G_j não contém caminhos de incremento;
3. $|f| = c(s,t)$ para algum corte (s,t) em G.

O tempo de execução do algoritmo de Ford-Fulkerson depende de como o caminho de incremento é determinado. Se ele for mal escolhido, o algoritmo poderá não terminar, pois, o fluxo poderá sofrer sucessivas ampliações (Cormen et al., 1998). No entanto, se o caminho de incremento for escolhido utilizando busca em largura, o algoritmo é executado em tempo polinomial.

Na prática, o tempo de execução do algoritmo de (Ford and Fulkerson, 1962) é da ordem de $O(E|f^*|)$, onde f^* é o fluxo máximo encontrado pelo algoritmo (Cormen et al., 1998).

Posteriormente, o método de (Ford and Fulkerson, 1962) foi incrementado por (Edmonds and Karp, 1972) de modo que a busca do caminho de incremento é a busca pelo caminho mais curto de *s* até *t* na rede residual. Com isso, o tempo de execução do algoritmo passou a ser polinomial na ordem de $O(VE^2)$

Dinic (Dinic, 1970) baseado nas idéias de (Edmonds and Karp, 1972), criou um método que procura o caminho de incremento mais curto, porém de uma forma mais eficiente, em tempo $O(|V|^2|E|)$. Essa melhora é alcançada porque o algoritmo de Dinic reutiliza as informações obtidas na busca em largura, que procura não apenas um, mas o número total de caminhos mais curtos. Por isso, enquanto procura pelo caminho de incremento mais curto de tamanho fixo l, o algoritmo mantém-se uma rede em camadas (*layered network*), que contém todos os caminhos mais curtos de *s* até *t*, de tamanho l. O tempo de construção da rede em camada é da ordem de $O(|E|)$ (Campos, 2009; Netto, 2003).

I.4.2 Algoritmo de Goldberg e Tarjan

O algoritmo de Goldberg (Goldberg and Tarjan, 1988) utiliza a idéia de pré-fluxo introduzido por (Karzanov, 1974) para computar os fluxos bloqueantes. O algoritmo de (Karzanov, 1974) mantém um pré-fluxo em uma rede não cíclica e empurra o fluxo através da rede para procurar os fluxos bloqueantes. Um pré-fluxo é semelhante a um fluxo, exceto pelo fato de que a quantidade de fluxo que entra em um

vértice pode ser menor que a quantidade que sai do vértice. Logo, em um pré-fluxo não tem-se a propriedade de conservação de fluxo. Um vetor g: $V \rightarrow R$ é chamado pré-fluxo se:

(i) $0 \leq g(u,v) \leq c(u,v) \; \forall \; (u,v) \in E$
(ii) $\sum_{(v,k)\in E} g(v,k) - \sum_{(u,v)\in E} g(u,v) \leq 0 \; \forall \; v \; \in \; V - \{1\}$

Para um pré-fluxo g, o grafo residual $R(g)$ e a capacidade residual são introduzidos analogamente ao do fluxo trocando as variáveis do fluxo por variáveis de pré-fluxo. Além disso, para cada vértice $v \in V$, define-se uma função excesso da seguinte forma:

$$e(v) = \begin{cases} \infty & \text{para } j = s \\ \sum_{(u,v)\in E} g(u,v) - \sum_{(v,k)\in E} g(v,k) & \text{para os demais casos.} \end{cases}$$

Dessa forma, um pré-fluxo é um fluxo se $e(v) = 0$ para todos os vértices $\in V - \{s,t\}$.

O algoritmo (Goldberg and Tarjan, 1988) abandona a idéia de procurar por um fluxo em cada fase e também a idéia de fase global. Ao invés disso, o algoritmo mantém um pré-fluxo na rede original e empurra o excesso de fluxo em direção ao sorvedouro que será o caminho mais curto no grafo residual. O excesso que não possa ser movido para o sorvedouro é retornado para a origem, também estimando o caminho mais curto. Apenas quando o algoritmo termina, o pré-fluxo torna-se um fluxo, e conseqüentemente o fluxo máximo.

Um outro ponto importante desse algoritmo é o cálculo da estimativa de tamanho do caminho de incremento mais curto da origem ao destino. Seja g um pré-fluxo. Um valor inteiro não negativo da função $d : V \longmapsto N$ é a função de distância se as seguintes condições forem atendidas:

(i) $d(n) = 0, \; d(j) > 0 \; \forall j \neq n$
(ii) $d(i) \leq d(j) + 1 \; para \; cada \; (i,j) \in R(g)$ tal que, $d(i) < n \; ou \; d(j) < n$

Dessa forma, um vértice j é dito ativo se $0 < d(j) < n$ e $e(j) > 0$.

O algoritmo 6 apresenta o algoritmo de Godberg (Goldberg and Tarjan, 1988), que é dividido em duas fases: a primeira determina um corte mínimo e a segunda transforma o resultado do pré-fluxo em um fluxo máximo.

Algoritmo 6: MAX-FLOW(V,E,s,t,c)

```
1  g ← 0
2  forall j ∈ V do
3      cn(j) ← first element of N(j)
4  end
5  Q ← {1}
6  while Q ≠ 0 do
7      select any active vertex j ∈ Q
8      Q ← Q - {j}
9      k ← cn(j)
10     while d(k) ≥ d(j) or res(j,k) > 0 do
11         if k = last element of N(j) then
12             RELABEL
13         else
14             cn(j) ← next element of N(j)
15         end
16         k ← j
17     end
18     PUSH
19     if j is active then
20         Q ← Q + {j}
21     end
22 end
23 PUSH
24 begin
25     δ ← min {e(j), res(j,k)}
26     if (j,k) ∈ A then
27         g(j,k) ← g(j,k) + δ
28     end
29     if (k,j) ∈ A then
30         g(k) ← g(k,j) - δ
31     end
32     if k is active then
33         Q ← Q + {k}
34     end
35 end
36 RELABEL
37 begin
38     cn(j) ← first element of N(j)
39     d(j) ← min{d(k) + 1 : (j,k) ∈ R(g)}
40 end
```

Esse algoritmo tem implementação seqüencial e paralela. O modelo seqüencial simples executa em tempo da ordem de $O(n^3)$ ou em tempo da ordem de $O(nm\ log(n^2/m)$ se for utilizado a estrutura de árvore dinâmica de Sleator e Tarjan (Sleator and Tarjan, 1981; Tarjan, 1983). A versão paralela do algoritmo tem tempo

de execução da ordem de $O(n^2\ logn)$ usando *n* processos e $O(1)$ palavra armazenada por aresta.

I.4.3 Algoritmo de fluxo máximo de (King et al., 1992)

O algoritmo de (King et al., 1992) é uma versão determinística do algoritmo aleatório de (Cheriyan et al., 1990), que reduz o problema de fluxo máximo à execução de uma estratégia em um jogo combinatório de dois jogadores onde a recompensa reflete o custo de computação. Nesse algoritmo, um fator aleatório é utilizado para escolher uma estratégia de sucesso. Em 1990 (Alon, 1990) encontrou o algoritmo mais rápido de fluxo máximo tirando a aleatoriedade da estratégia de (Cheriyan and Hagerup, 1989) com recompensa a um custo na ordem de $n^{2/3}/\log n$, onde n é o número de vértices. Nesse ponto, o algoritmo de (King et al., 1992), modificando algumas regras do jogo é capaz de encontrar uma estratégia determinística melhorando o custo da recompensa para um fator n^{ϵ} da versão aleatória.

Cheriyan et al.(1990) reduziu o custo de encontrar um fluxo máximo para $O(mn+ n^{3/2}m^{1/2} + P(n^2, nm)\log n + C(n^2, mn))$ onde $P(n, m)$ é o número de pontos que podem ser obtidos pelo adversário no jogo e $C(n, m)$ o custo de implementação da estratégia do algoritmo. Seja $G = (U, V, E)$ um grafo bipartido não direcionado com $|V| = |U| = n$ e $|E| = m$, onde U e V são os vértices e E as arestas do grafo.

O primeiro jogador é o algoritmo, que para cada vértice u $\in$ U ordena as arestas adjacentes a u. A medida que o jogo prossegue, a maior aresta remanescente a cada vértice $u \in U$ ainda no grafo é chamada aresta " designada". Os movimentos remanescentes são executados pelo adversário que pode remover um vértice $v \in V$ e todas as suas arestas do grafo, ganhando um ponto para cada aresta adjacente a v, ou remover apenas uma aresta, mas sem pontuar pelo movimento. A diferença do algoritmo de (King et al., 1992) para o algoritmo de (Cheriyan et al., 1990) é que esse permite ao algoritmo atuar adaptativamente e também permite que uma aresta seja redesignada (King et al., 1992).

As etapas do jogo são: inicialmente o algoritmo designa uma aresta para cada vértice em U, de modo que o jogo prossegue até que todos os vértices $\in V$ tenham sido removidos.

O adversário executa um dos seguintes movimentos:

- Remover qualquer aresta do grafo. Quando o adversário remove uma aresta designada nenhum ponto é computado por esse movimento;
- Remover qualquer vértice $\in V$ e suas arestas adjacentes pontuando um ponto para cada aresta designada.

O algoritmo pode executar qualquer seqüência dos seguintes movimentos:

- Pode designar uma aresta para cada vértice $u \in U$ que não esteja designada para um vértice em V;
- Pode designar uma aresta, isto é, ele pode mudar uma aresta designada incidente a um vértice $u \in U$ para outro, nesse caso, o adversário ganha um ponto para cada redesignação.

Dessa forma, o tempo de execução do algoritmo de (King et al., 1992) é da ordem de $O(mn + n^{2+\epsilon})$ para qualquer constante ϵ.

I.4.4 Algoritmo de fluxo máximo de (Ahuja and Orlin, 1989)

Uma operação limitante do algoritmo de (Goldberg and Tarjan, 1988) é o número de *pushes* não saturados que é $O(n^3)$. Esse tempo computacional é reduzido para $O(nm\ \log(n^2/m))$ através da utilização de estrutura de árvore dinâmica (Ahuja and Orlin, 1989). No entanto, (Ahuja and Orlin, 1989) mostraram que o número de *pushes* não saturados pode ser reduzido para $O(n^2\ \log U)$ utilizando o conceito de *excess scaling*, modificando o algoritmo de (Goldberg and Tarjan, 1988) sem utilizar nenhum estrutura de dados especial, como árvore dinâmica. Essa característica, segundo (Ahuja and Orlin, 1989) torna o algoritmo mais fácil de implementar e mais eficiente na prática, porque requer apenas estrutura de dados simples, com baixo *overhead* computacional.

Scaling foi introduzido por (Edmonds and Karp, 1972) para resolver o problema do fluxo de custo mínimo e mais tarde estendido por (Gabow, 1985) para outros problemas de otimização de rede. No caso do problema de fluxo máximo, *scaling* transforma iterativamente um problema em outro, porém de menor magnitude. Por exemplo, um número k torna-se $\lceil k/2 \rceil$. A solução do problema menor é então próxima de ótimo da solução do problema anterior (Ruhe, 1991).

O algoritmo de (Ahuja and Orlin, 1989) executa $\log U$ iterações de *scaling* cada uma requerendo $O(n^2)$ *pushes* não saturados, onde U é o número de vértices do grafo. Desse modo, se um fluxo é empurrado para um vértice com excesso suficientemente grande para outro com excesso suficientemente pequeno, não permitindo que o excesso fique muito grande, o tempo computacional do algoritmo é da ordem de $O(nm + n^2 \log U)$. Em redes não densas e não esparsas, o algoritmo executa em tempo $O(nm)$ o que melhora o tempo do algoritmo de (Goldberg and Tarjan, 1988) (Ahuja and Orlin, 1989; Ruhe, 1991).

O algoritmo 7 apresenta o algoritmo de (Ahuja and Orlin, 1989)

Algoritmo 7: MAX-FLOW

```
1  PREPROCESS
2  K ← 1 + ⌈logU⌉
3  Para k=1 to K Faça
4      Δ = 2^(K−k)
5      Para cada i ∈ N faça
6          se e_i > Δ/2 então
7              adicione i a LIST(d(i))
8          fim se
9      fim
10     level ← 1
11     enquanto level < 2n faça
12         se LIST(level) = 0 então
13             level ← level + 1
14         senão
15             selecione um vértice i da LIST(level)
16             PUSH/RELABEL(i)
17         fim se
18     fim enquanto
19 fim para
```

Algoritmo 8: PUSH/RELABEL(i)

```
1  achou ← falso
2  seja(u,v) a aresta atual do vértice v
3  enquanto achou = false E (u,v) ≠ null faça
4      se d(i) = d(j) + 1 E r_uv > 0 então
5          achou ← verdadeiro
6      senão
7          troque a aresta atual do vértice i pelo próximo arc(u,v)
8      fim se
9  fim enquanto
10 se achou então
11     push min {e_u, r_u,v, Δ − e_v} unidades do fluxo na aresta (u,v)
12     atualize a capacidade residual r_uv e o excesso e_u e e_v
13     se excesso atualizado e_u ≤ Δ/2 então
14         exclua o vértice u da LIST(d(i))
15     fim se
16     se j ≠ s ou t E (excesso atualizado e_e > Δ/2) então
17         adicione o vértice j à LIST(d(j))
18         level ← level -1
19     fim se
20 senão
21     exclua o vértice i da lista LIST(d(i))
22     atualize d(i) ← min{d(j) + 1 (i,j) ∈ A(i) E r_ij > 0}
23     adicione o vértice i a LIST(d(i)) E atribua a aresta atual do vértice i para o
       primeiro vértice de A(i)
24 fim se
```