Descoberta de Recursos em Grades Computacionais Utilizando Estruturas P2P

Vladimir Rocha , Marco A. S. Netto , Fabio Kon

[1]Departamento de Ciência da Computação
Instituto de Matemática e Estatística
Universidade de São Paulo.
Rua do Matão, 1010 – 05508-090 São Paulo, SP

{vmoreira, netto, kon}@ime.usp.br

Abstract. *Recent research in Grid systems have focused on efficient resource discovery using P2P techniques. In this paper, we describe two novel strategies to discover resources in Computational Grids. The first strategy, based in domain neighborhood, makes use of information from network layer to find resources in closed domains regarding latency. The second strategy, based on resources spread across different domains, a new structure enables an efficient search for resources in range values. This structure can be used in more real environments than existing solutions since it allows the storage of resources with same value, and it is also more simple to be managed.*

Resumo. *Pesquisas recentes mostram uma tendência na utilização de técnicas P2P para a localização de recursos em Grades Computacionais. Neste artigo apresentamos duas novas estratégias para a busca de recursos em Grades. Na primeira estratégia, baseada na vizinhança de um domínio, utilizamos informações mantidas pela camada de rede para localizar os recursos dos domínios mais próximos em termos de latência. Na segunda estratégia, baseada nos recursos localizados em domínios dispersos globalmente, criamos uma nova estrutura que permite a busca eficiente por recursos com valores presentes dentro de um determinado intervalo. Essa estrutura pode ser utilizada em ambientes mais reais que as soluções existente pois ela permite o armazenamento de recursos com o mesmo valor, além de ser mais simples de administrar.*

1. Introdução

Os sistemas de Computação em Grade surgiram nos anos 90 para atender a necessidade do compartilhamento de recursos para a execução de uma tarefa distribuída que utilize esses recursos, como processamento de alto desempenho, manipulação de grandes volumes de dados, simulações, entre outras. Como em uma Grade Computacional os recursos estão geograficamente distribuídos, a localização de tais recursos torna-se uma tarefa bastante complexa, especialmente quando há uma grande quantidade de recursos envolvidos [Iamnitchi et al. 2002]. Portanto, protocolos eficientes para o compartilhamento de recursos entre diferentes domínios tornou-se uma grande necessidade para os usuários das Grades.

Com a mesma necessidade de compartilhar recursos temos as redes *Peer-to-Peer* (P2P ou Par-a-Par) [Nelson Minar et al. 2001], inicialmente projetadas com o objetivo do

compartilhar arquivos entre milhares de usuários. Nestes ambientes, dois são os requisitos fundamentais para seu funcionamento: (i) ser dinâmico pois os usuários podem entrar e sair da rede de forma intermitente; (ii) ser escalável para que milhares de usuários conectados possam compartilhar recursos.

Existem diversas propostas para a localização de recursos em redes P2P. As primeiras propostas foram baseadas em propagação de mensagens de busca [Lv et al. 2002]. As propostas mais recentes são baseadas em estruturas distribuídas que utilizam algoritmos eficientes para o roteamento de mensagens, evitando assim sua propagação descontrolada [Stoica et al. 2001, Zhang and Schopf. 2004, Ratnasamy et al. 2001].

Nos últimos anos, pesquisadores da área de Grades têm se concentrado em utilizar as técnicas de localização e descoberta de recursos nas redes P2P, como *"flooding"*, réplicas e propagação controlada [Iamnitchi et al. 2002].

Neste artigo, propomos duas estratégias de localização de recursos em Grades Computacionais. A primeira busca os recursos na vizinhança de um domínio e a segunda busca os recursos em domínios globalmente dispersos. No caso da vizinhança, utilizamos informações mantidas pela camada de rede para localizar os recursos dos domínios mais próximos em termos de latência. No caso dos domínios globalmente dispersos, uma nova estrutura, mais simples de administrar que as atuais, foi desenvolvida para permitir a busca eficiente por esses recursos. Além disso, ao contrário das soluções existentes, descrevemos uma estrutura de dados que trata as repetições de valores durante uma busca por recursos. Nas abordagens apresentadas, utilizamos uma estrutura freqüentemente encontradas nas redes P2P chamada de Tabela de *Hash* Distribuída.

O restante deste artigo está organizado da seguinte maneira. Na Seção 2 apresentamos como é feita a localização dos recurso nas Grades Computacionais e nas redes P2P. Nas Seções 3 e 4 são descritas as estratégias para o problema de busca de recursos na vizinhança e nos domínios globalmente dispersos, respectivamente. Na Seção 5 apresentamos resultados dos experimentos realizados em um simulador para avaliar as soluções propostas. Na Seção 6 apresentamos os trabalhos relacionados e, finalmente, na Seção 7 apresentamos as conclusões e trabalhos futuros.

2. Descoberta de Recursos

A descoberta eficiente de recursos é um serviço fundamental para sistemas baseados em compartilhamento de recursos, como Grades Computacionais e redes P2P. Neste artigo, vamos nos concentrar na busca de recursos dinâmicos, que são aqueles onde algum dos seus atributos muda de valor freqüentemente. A quantidade de memória RAM disponível, percentual livre do processador ou o espaço livre em disco rígido de um computador são exemplos de recursos cuja disponibilidade varia dinamicamente.

2.1. Grades Computacionais

Os sistemas de Grade Computacional [Foster and Kesselman 2003] podem ser definidos como infra-estruturas de software que permitem coordenar, interligar e compartilhar recursos heterogêneos, possivelmente distribuídos geograficamente.

A arquitetura típica de uma Grade [de Camargo et al. 2004] é composta por nós e gerenciadores. Os nós são os responsáveis por compartilhar recursos e os gerenciadores

são os responsáveis pela administração, coleta de informações e escalonamento de tarefas. Na Figura 1 mostramos a interligação entre os diferentes domínios administrativos de uma Grade, ou seja, dos gerenciadores junto com os nós administrados por eles.

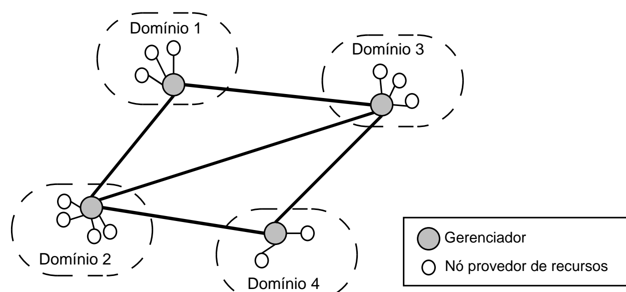


Figura 1. Arquitetura típica de uma Grade.

2.1.1. Busca de Recursos

As alternativas atuais para localização de recursos em Grades Computacionais, sem considerar a utilização de estruturas P2P, são baseadas em servidores centralizados ou hierarquias estáticas. As soluções de busca baseadas em servidores centralizados [Fitzgerald et al. 1997] têm se mostrado pouco eficientes para sistemas que compartilham recursos dinâmicos dispersos geograficamente. Algumas pesquisas [Cosway 1997a] conseguiram melhorar essa eficiência utilizando réplicas dos recursos. Todavia, administrar tais réplicas ainda gera alta sobrecarga na rede devido à quantidade de mensagens trocadas para mantê-las consistentes [Basu et al. 2005].

As soluções com hierarquias estáticas [Fitzgerald 2001] apresentam melhorias em relação às alternativas que utilizam servidores centralizados, onde a estrutura da hierarquia é distribuída nos nós pertencentes à Grade. Porém, esta alternativa é ineficiente na busca de recursos dinâmicos, devido à natureza estática da hierarquia. Outro problema das soluções que utilizam hierarquias é a alta sobrecarga que podem ter alguns nós em relação aos outros, resultando em problemas de escalabilidade [Adjie-Winoto et al. 1999].

2.2. Redes P2P

A maioria dos sistemas P2P tem como objetivo a busca e transferência de documentos. Entre as características que uma rede P2P pode ter encontram-se [Leuf 2002]: a comunicação é direta entre os pares, sem presença de um mediador central; é altamente escalável e a conectividade dos pares é usualmente transitória e não permanente.

Existem vários modelos arquiteturais para redes P2P [Leuf 2002]. A seguir serão mostradas as características de três delas, as quais utilizaremos nas soluções apresentadas nas Seções 3 e 4.

Modelo Atômico. O modelo atômico é chamado pelos puristas de "a verdadeira rede P2P". As características deste modelo são as seguintes: (1) Não existem servidores centrais que possam ter informação completa da rede e dos seus usuários. (2) A estrutura de conexão formada pelos nós é aleatória, uma vez que um nó pode se conectar potencialmente com qualquer outro nó. (3) Cada nó é autônomo e administra seus próprios

recursos e suas próprias conexões. (4) Para o ingresso de um nó na rede é necessário conhecer pelo menos um outro nó ao qual se conectar.

Modelo Estruturado. Este modelo foi um dos últimos a surgir no cenário dos modelos P2P. A idéia principal é a de construir redes estruturadas de modo a melhorar a eficiência das buscas por recursos nos nós da rede. O modelo estruturado tenta evitar que a busca por um recurso seja propagada sem controle em nós que não têm o recurso requisitado, como acontece no modelo atômico. Existem diferentes estruturas, segundo o modelo estruturado, criadas para armazenar e procurar recursos em redes P2P, tais como: listas distribuídas [Aspnes and Shah 2003], árvores distribuídas [Cosway 1997b] e tabelas de *hash* distribuídas [Stoica et al. 2001].

Tabela de *Hash* Distribuída. A Tabela de *Hash* Distribuída (DHT) [Tanenbaum 2002] é uma estrutura que permite a realização das funções de uma tabela de *hash* normal. Nela pode-se armazenar um par $(chave, valor)$ e procurar por esse valor utilizando a $chave$. Uma das características importantes sobre as DHTs é que o armazenamento e as operações de busca são eficientes, escaláveis e distribuídas entre as máquinas que pertencem à rede P2P [Rowstron and Druschel 2001]. Nesta estrutura, cada recurso armazenado recebe um identificador único ID, que é mapeado a um par responsável por ele.

2.2.1. Busca de Recursos

Nos sistemas P2P, cada modelo arquitetural apresentado na seção anterior possui soluções diferentes para uma busca por recursos.

A busca no modelo atômico é baseada na propagação de mensagens através dos nós da rede. Uma das primeiras soluções utilizadas era propagar a requisição de busca a "todos" os pares da rede (*flooding*). Uma variação desta alternativa utiliza uma variável TTL (*Time To Live*) que limita a propagação da mensagem. O valor desta variável diminui a cada propagação, controlando assim a quantidade de mensagens que utilizam a rede. A desvantagem se produz quando a quantidade de pares que utilizam a rede aumenta. Neste caso, a quantidade de mensagens aumenta exponencialmente deixando a rede inutilizada rapidamente pela sobrecarga [Cohen and Shenker 2002]. Outras alternativas mais recentes, como a do protocolo do Gnutella 2[1], propõem que um par p (conhecido como *super par ou hub peer*) armazene os nomes dos recursos dos seus conhecidos. Assim, quando uma mensagem de busca chega a p, ele a encaminha diretamente aos pares responsáveis.

No modelo estruturado, devido à infra-estrutura de conexão dos pares, a procura por recursos sempre devolve um resultado positivo caso o recurso exista. O caminho seguido pela mensagem para procurar o par que armazena o recurso é o mais eficiente possível – evitando a propagação de mensagens do modelo atômico. É importante destacar que todas essas estruturas utilizam $O(\log n)$ troca de mensagens para localizar o recurso (onde n é a quantidade de pares da rede). Por sua vez, as DHTs provêm uma análise matemática e simulações deste comportamento [Stoica et al. 2001, Gummadi et al. 2003].

[1] O protocolo do Gnutella 2 pode ser encontrada em: http://www.gnutella2.com.

3. Busca por Recursos na Vizinhança

Quando um nó da Grade procura por recursos, é interessante obtê-los dos domínios que estejam mais próximos do nó solicitante em termos de latência. Com isso, a comunicação entre o requisitante e o responsável pelo recurso será mais eficiente, melhorando assim o desempenho da execução da tarefa que utiliza esse recurso.

Esta estratégia, baseada no modelo atômico, visa interligar os gerenciadores dos domínios mais próximos entre si. No modelo atômico, a estrutura e formação dos gerenciadores é criada de forma aleatória em relação a seus vizinhos[2].

3.1. Usando e Armazenando a Informação dos Roteadores

Para resolver o problema da busca de recursos na vizinhança, utilizamos a função de roteamento da camada de rede. A camada de rede tem como função entregar pacotes de um lugar a outro através de uma infra-estrutura de redes interconectadas [Stallings 2003]. Para isso, os protocolos usados nesta camada determinam os caminhos entre destinos, permitindo assim estabelecer a rota de preferência para o envio de pacotes.

A compreensão da rota seguida por um pacote pode ajudar a descobrir que domínios estão próximos em termos de latência. Conforme ilustrado na Figura 2, se um gerenciador $g1$ envia uma mensagem ao gerenciador $g2$, o pacote enviado segue uma rota $r1$ determinada pela camada de rede. Suponhamos que em um dos roteadores dessa rota exista outra rota $r2$ para o gerenciador $g3$, que não é conhecido e que está mais próximo de $g1$, em termos de latência, do que $g2$. Se pudéssemos conhecer $g3$, $g1$ poderia se comunicar com $g3$, assim, a comunicação entre eles provavelmente seria mais eficiente do que a comunicação entre $g1$ e $g2$.

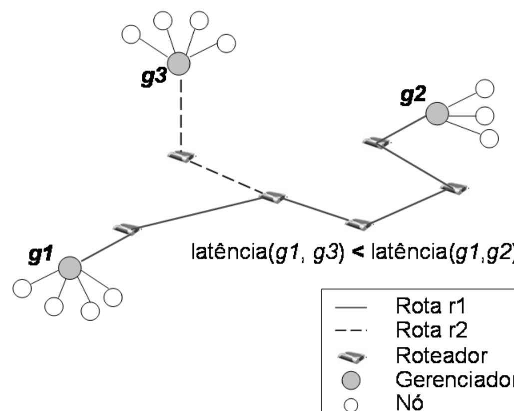


Figura 2. Rota da mensagem entre dois gerenciadores.

Nossa estratégia armazena e utiliza as informações de latência e das rotas da seguinte maneira. *Objetos roteadores*, mantidos no nível do middleware, mantêm informações sobre cada roteador presente em uma rota entre dois gerenciadores. A estrutura destes objetos corresponde à latência e ao identificador do gerenciador mais próximo a ele. Para o armazenamento e recuperação eficiente desses objetos, utilizamos uma Tabela de *Hash* Distribuída.

[2]A vizinhança de um gerenciador g é definida como os gerenciadores que g conhece.

3.2. Cache de Gerenciadores

Para facilitar o processo de ingresso de um gerenciador na Grade e a busca de recursos na vizinhança, foi criada um *cache*, que acelera a obtenção dos vizinhos de um domínio. Esse cache corresponde a uma lista de identificadores[3] dos gerenciadores, ordenada em ordem crescente de latência, armazenada localmente em cada gerenciador e que é atualizada periodicamente.

3.3. Ingresso de um Gerenciador na Grade

A estratégia de ingresso de um gerenciador de domínio $g1$ à Grade é representada pelo algoritmo da Figura 3. O gerenciador que ingressa na Grade precisa conhecer pelo menos um gerenciador onde se conectar. Se for a primeira vez (Linha 3), esses gerenciadores podem se obtidos de uma fonte na Internet, como por exemplo, uma página Web. Se, antes do ingresso, o $g1$ já pertencia à Grade, os gerenciadores podem ser obtidos do seu cache (Linha 5). Com o mais próximo dos gerenciadores encontrados (Linha 6), usamos os recursos físicos de rede que permitem encontrar o melhor caminho entre $g1$ e o $escolhido$. Para isto, obtemos os endereços IP dos roteadores do caminho entre o $g1$ e o escolhido (Linha 7). A obtenção desses roteadores pode ser feita, por exemplo, utilizando o comando *traceroute* do Unix [Stallings 2003]. Para todos os endereços IP dos roteadores encontrados, consultamos na DHT se existe alguma informação (latência e identificador do gerenciador mais próximo) sobre esses roteadores (Linha 8). A seguir, obtém-se o gerenciador mais próximo com o qual $g1$ deveria conectar-se (Linhas 10 a 14) fazendo comparações de latência. Finalmente, o algoritmo devolve o $escolhido$ com o qual $g1$ se conectará (Linha 16).

```
1: INGRESSO (Gerenciador g1)
2:   se (primeiro ingresso)
3:       possíveis ← obter_gerenciadores_da_Internet()
4:   caso contrário
5:       possíveis ← obter_gerenciadores_do_cache()
6:   escolhido ← obter_gerenciador_menor_latência(possíveis)
7:   ip_roteadores ← roteadores_entre_g1_e_escolhido()
8:   objetos_roteador ← dht.roteadores_entre_g1_e_escolhido(ip_roteadores)
9:   para cada roteador em objetos_roteador {
10:      g2 ← roteador.gerenciador_mais_próximo()
11:      se (latência(g1, g2) < latência(g1, escolhido))
12:          escolhido ← g2
13:      caso contrário
14:          dht.atualiza_roteador(roteador, escolhido)
15:  }
16:  devolve escolhido
```

Figura 3. Algoritmo para o ingresso de um gerenciador na Grade.

3.4. Atualização das Estruturas Internas

Nossa estratégia requer que as estruturas internas tenham as referências aos gerenciadores o mais atualizadas possível. As atualizações são feitas periodicamente por processos e independente de um gerenciador entrar ou sair da Grade. Existem três tipos de atualizações. As duas primeiras são as que dizem respeito à estrutura lógica das conexões e a terceira corresponde à atualização do cache, ou seja, à forma de obter novos gerenciadores.

[3] O identificador deve ser único, por exemplo, pode-se usar o endereço IP e a porta TCP do gerenciador.

Atualização do escolhido. A atualização do escolhido pode ser feita quando um gerenciador entra ou sai da Grade. Caso um gerenciador g tenha como referência o $escolhido$, g é responsável por verificar, periodicamente, se o escolhido está acessível. Caso não esteja, g deve se conectar com o primeiro gerenciador do seu cache local. Se não existir referência alguma no cache, então ele deve executar o algoritmo para conectar-se à Grade da Figura 3.

Atualização de objetos roteadores. A atualização dos objetos roteadores ocorre quando um gerenciador se desconecta da Grade ou para evitar a sobrecarga do gerenciador que está no objeto roteador. Em qualquer dos dois casos, é necessário remover a referência contida no objeto roteador. Para isso, cada objeto roteador verifica se o gerenciador armazenado não está acessível ou se o limite de tempo de permanência foi atingido, removendo o objeto roteador da DHT.

Atualização do cache. Essa atualização consiste em: (a) eliminar as referências aos gerenciadores do cache que não estejam disponíveis e (b) obter novos gerenciadores que estejam mais próximos em termos de latência. No primeiro caso, o processo verifica, com um teste de conexão, se os gerenciadores do cache estão acessíveis ou não. Para isso, tem-se que enviar e receber mensagens de todos os gerenciadores do cache, o que pode resultar em uma saturação da largura de banda para aplicações com centenas de milhares de gerenciadores. Para evitar a saturação, uma alternativa é através da atualização das primeiras entradas do cache. No segundo caso, ou seja, na obtenção de novos gerenciadores, o processo pergunta a algum dos vizinhos tomado de forma aleatória se ele conhece outros gerenciadores. Esses gerenciadores, que não existem no cache, devem cumprir com o requisito de estarem entre as primeiras T posições do cache em termos de latência, onde T é um valor determinado pelo usuário. No caso do gerenciador g ficar na primeira posição, g será o novo $escolhido$. A mensagem de busca é propagada com uma profundidade determinada pelo valor de uma variável TTL (*Time to Live*) decrementada a cada propagação. Para evitar a geração de ciclos, ou seja, a pergunta ser refeita para um mesmo gerenciador, pode-se enviar na mensagem a lista dos gerenciadores que já foram visitados. Uma outra alternativa é deixar que ciclos ocorram e o gerenciador que fez a requisição descarte os gerenciadores que são iguais.

4. Busca Global por Recursos

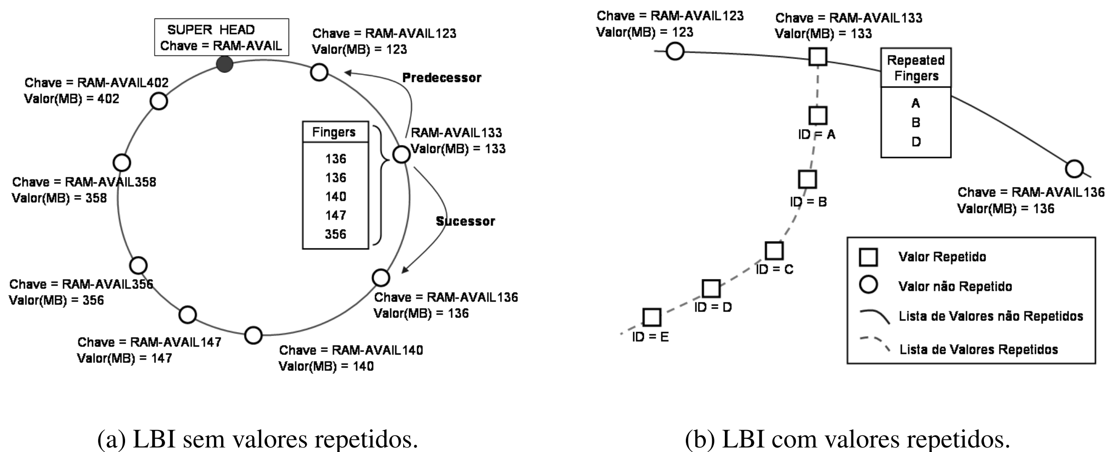
A busca por recursos dinâmicos em domínios globalmente dispersos é feita em uma lista distribuída entre os nós da Grade. Essa lista contém referências aos recursos disponibilizados por eles. Para o armazenamento das informações sobre os recursos a serem localizados, utilizamos a Tabela de *Hash* Distribuída (DHT), uma estrutura muito eficiente na busca de recursos. Entretanto, uma limitação das DHTs é que elas não foram desenvolvidas para permitir buscas por intervalo de identificadores, onde um intervalo é definido pelos valores que estão entre um limite inferior e superior. Um exemplo de busca por intervalo é encontrar 64 máquinas que tenham um mínimo de 256 MB de RAM disponível. Para resolver esse tipo de problema, estendemos a lista distribuída, que denominamos de Lista para Busca por Intervalo, descrita a seguir.

4.1. Estrutura Simplificada

A estrutura, que denominamos Lista para Busca por Intervalo (LBI), é uma lista distribuída, ordenada por um valor e que permite buscas por intervalo. As buscas e

inserções de recursos, dado um identificador, utilizam $O(\log n)$ trocas de mensagens, onde n corresponde à quantidade de recursos armazenados na LBI. Esses recursos são indexados por um valor e inseridos na posição correta da LBI de modo que a lista esteja sempre ordenada de forma crescente.

Cada recurso armazenado possui um identificador do nó que compartilha o recurso (como o endereço IP), o valor do recurso, o identificador do gerenciador do domínio do nó e a chave do recurso, definida como a união do nome do recurso e o valor, por exemplo, RAM-AVAIL64 corresponde ao nó com RAM disponível de 64 MBytes. As estruturas internas de cada recurso são ponteiros para seu predecessor, seu sucessor e uma tabela de ponteiros – também conhecida como tabela de *fingers* ou tabela de roteamento – para outros recursos, usada para acelerar as buscas. Na Figura 4(a), mostramos a estrutura com oito recursos e identificadores não repetidos. Note que se quisermos compartilhar, por exemplo informação sobre o uso de memória e processador, teremos que ter uma LBI para cada tipo de recurso compartilhado, e fazer uma filtragem sobre as LBIs. Detalhes de como é feito esse processo podem ser encontrados em [Rocha 2005].



(a) LBI sem valores repetidos.

(b) LBI com valores repetidos.

Figura 4. Estrutura LBI sem e com valores repetidos.

A tabela de *fingers* de um recurso r é um conjunto de identificadores onde cada registro i (r e i são inteiros positivos) dessa tabela corresponde ao recurso que sucede r com um valor maior ou igual a $r + 2^{i-1}$ [Stoica et al. 2001]. Por sua vez, a LBI possui uma cabeça de lista chamada *super head* que mantém o nome do recurso armazenado, seu sucessor e a tabela de *fingers*.

4.2. Estrutura Estendida

Em sistemas reais, é bastante comum encontrar situações em que a busca por um valor, dado um identificador, tenha como resposta várias saídas. Por exemplo, supondo que o identificador é pessoa e o valor é idade, deseja-se buscar todas as pessoas com 17 anos de idade. Até o momento, as soluções existentes não demonstram como tratar as repetições dos valores.

Como mostra a Figura 4(b), para comportar os casos onde existe repetição de valores, a estrutura simplificada foi estendida de duas maneiras: (1) recursos com valores

repetidos são adicionados de modo a formar uma nova lista ligada a partir da original e (2) uma outra tabela, que denominamos (*repeated fingers*), é inserida para permitir que em uma busca sejam devolvidos os recursos com valores repetidos com troca de mensagens em $O(\log n)$, como descrito na Seção 4.3. Sem essa tabela, a devolução de todos os valores é de ordem linear.

Na lista para valores repetidos, os recursos não estão ordenados de forma alguma. Ela somente é criada se o valor é repetido e é removida se ela não contém mais recursos repetidos. Finalmente, a tabela de *repeated fingers* para um recurso r contém as entradas i (partindo de zero) que representam recursos s que estão a uma distância (quantidade de recursos entre r e s na lista de valores repetidos) igual a 2^i. Na Figura 4(b), vemos que o recurso com valor 133 tem em sua tabela de *repeated fingers* o nó com ID = D, que representa o recurso que está a uma distância (do recurso com valor 133) igual a 4 = 2^i, pois a posição de D nesta tabela é $i = 2$.

4.3. Buscas por Intervalo

Para buscar recursos pertencentes a um intervalo $[i, e]$ utilizamos o algoritmo ilustrado na Figura 5. O primeiro passo antes de se chamar a função SEARCH é obter a cabeça da lista de identificadores chamada *super_head*. Em posse deste recurso, o requisitor da busca chama a função SEARCH(i, e, $super_head$) que devolverá os recursos contidos no intervalo $[i, e]$. A busca por intervalo requer $O(\log n)+m$ troca de mensagens, onde m é o número de recursos devolvidos.

1: **SEARCH(Intervalo** $[i, e]$**, Recurso** r**)**
2: lista ← ∅
3: valores ← r.obter_valores_por_intervalo(i, e)
4: **para cada** valor v **em** valores
5:     $recurso$ ← v.recurso
6:     lista ← lista ∪ **SEARCH**([v, e], $recurso$)
7: **devolve** lista

Figura 5. Algoritmo para buscar os valores contidos em um intervalo.

Depois de obter do recurso r os valores contidos no intervalo $[i, e]$ (Linha 3), ocorre uma iteração sobre esses valores para obter novos valores (Linhas 4-6). Finalmente, o método SEARCH devolve uma lista com todos os recursos encontrados (Linha 7). O método $obter_valores_por_intervalo$ (Linha 3) devolve todos os valores no intervalo $[i, e]$, com seus respectivos recursos, mantidos nas tabelas *fingers* e *repeated fingers* de r.

Como o foco deste trabalho são as buscas por recursos, não são discutidos aqui os processos de inserção, atualização e eliminação deles, bem como a atualização da LBI. A descrição detalhada desses processos podem ser encontrados em [Rocha 2005].

4.4. Buscas de Recursos com Requisitos Compostos

Como apresentado na Seção 4.1, a LBI armazena informações para um tipo específico de recurso. No caso de querermos realizar uma busca por recursos que satisfaçam um conjunto de requisitos, por exemplo, disponibilidade de memória > 512, disponibilidade de processador $> 90\%$, teremos que usar uma LBI para cada tipo de requisito. Neste cenário, a busca por recursos é efetuada da seguinte maneira: (1) para cada tipo de requisito, através do algoritmo de busca da seção anterior, são obtidos da sua respectiva LBI

os recursos que atendem ao critério da busca; (2) é feito um emparelhamento entre os nós das listas obtidas e devolvido ao requisitante somente os recursos que estão contidos em todas as listas. Se um nó não pertencer a uma das listas quer dizer que não satisfaz todos os requisitos da busca.

5. Resultados Experimentais

Nesta seção, avaliaremos as duas estratégias aqui propostas através de simulações. Para realizar os experimentos, implementamos nossas estratégias em Java, usando Bamboo[4] como a implementação da DHT que armazenará a LBI, e o seu simulador. Adaptamos o simulador para estimar, a partir da troca de mensagens, a largura de banda utilizada. A plataforma de simulação foi um PC com um processador Intel de 2.4 GHz, 2 GBytes de RAM e sistema operacional GNU/Linux.

Para ambas estratégias, utilizamos a topologia de rede baseada no King [Gummadi et al. 2002]. Esta topologia, muito utilizada em simulações, representa uma situação realista de uma grande variedade de *hosts* da Internet, com suas restrições de largura de banda e latência. No caso do Bamboo, os parâmetros como o tempo de atualização da tabela de *fingers*, mensagens *keep-alive*, entre outros, utilizamos os que estão definidos pelo próprio Bamboo e que podem ser encontrados em [Rhea et al. 2004].

5.1. Busca na Vizinhança

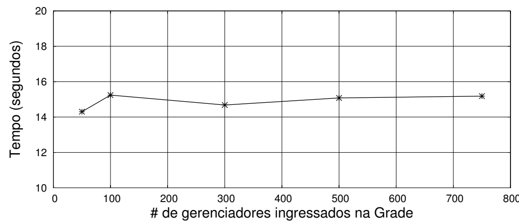
Na busca por recursos na vizinhança usamos um modelo de rede onde existem dois tipos de nós: os roteadores e os gerenciadores. Os nós gerenciadores somente podem se unir aos nós roteadores e um nó roteador deve estar unido a outros roteadores ou gerenciadores. Para efeitos de simplicidade, a comunicação entre os nós é simétrica. Ou seja, se um nó a se comunica com um nó b, com uma latência l, então b se comunica com a com a mesma latência. Os parâmetros utilizados para as atualizações das estruturas internas foram: 5 minutos a cada atualização do *escolhido* e 10 minutos para cada atualização dos *objetos roteadores* e do *cache*.

O custo de ingresso na Grade de um gerenciador, em função da quantidade de gerenciadores que já ingressaram, é muito importante para verificar a escalabilidade as estratégias. Neste experimento usamos de 50 até 750 gerenciadores e 300 roteadores. Como vimos na Seção 3.3, para ingressar na Grade, o gerenciador só depende de uma conexão lógica (referência) com um outro. A Figura 6(a) mostra que o custo para encontrar a referência ao $escolhido$ é praticamente constante.

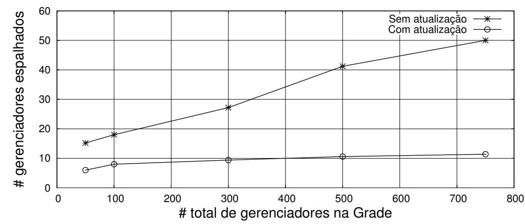
Quando um gerenciador ingressa pela primeira vez na Grade lhe é dado, de forma aleatória, vários gerenciadores aos quais se conectar. Nesse caso, existe uma probabilidade que sua vizinhança esteja distante em termos de latência. Definimos então, como *gerenciador espalhado*, o gerenciador onde a maior parte da sua vizinhança são gerenciadores distantes. Nesta simulação, definimos como espalhado o gerenciador cujo cache tem 50% das referências com latência maior a 200 mseg. Na Seção 3, mostramos que a vizinhança é modificada no processo de atualização do cache.

A quantidade de gerenciadores espalhados é mostrada na Figura 6. Podemos observar que, quando não utilizado o processo de atualização do cache, existem muitos

[4]Página oficial do projeto Bamboo: http://www.bamboo-dht.org.



(a) Custo de ingresso na Grade.

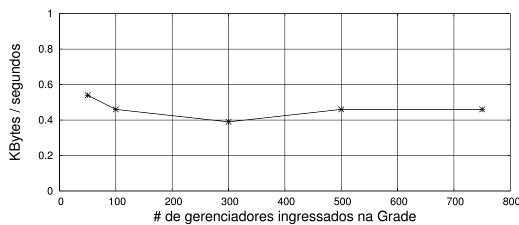


(b) Gerenciadores espalhados no ingresso na Grade.

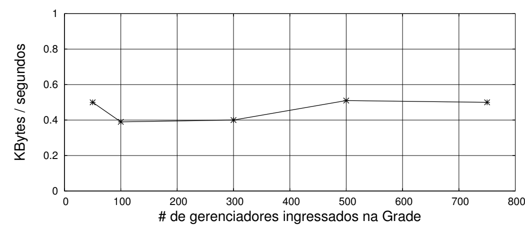
Figura 6. Gráficos de simulação da alternativa da vizinhança.

gerenciadores espalhados. Entretanto, essa quantidade diminui quando a atualização do cache é executada pois neste processo tenta-se obter uma vizinhança que esteja mais perto (ver Seção 3.4). Por exemplo, podemos observar que em uma rede com 500 nós, 40 deles são espalhados, mas este valor diminui para 10 quando executada a atualização do cache.

O *consumo da largura de banda usada* por uma aplicação é um tema muito importante em recentes pesquisas que testam a escalabilidade. Podemos observar na Figura 7(a) que o consumo de largura de banda ***por gerenciador***, quando executado o ingresso dele na Grade é aproximadamente constante. Em nossas simulações observamos que o tempo necessário para esse ingresso é aproximadamente 13 segundos, e portanto, a quantidade total de dados transferidos é de aproximadamente 5 KBytes.



(a) Largura de banda usada na entrada de um gerenciador na Grade.



(b) Largura de banda usada na atualização do cache.

Figura 7. Consumo de largura de banda utilizada na entrada de um gerenciador na Grade e na atualização do seu cache.

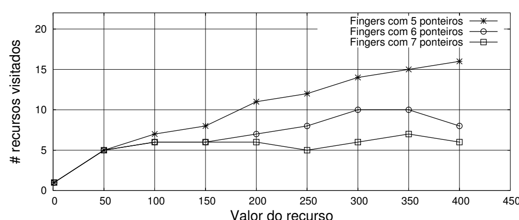
No caso da atualização do cache, o consumo da largura é aproximadamente constante (Figura 7(b)). Como apresentado na Seção 3.4, isso se deve ao fato do cache fazer requisições somente a uma quantidade limitada de vizinhos (em nosso caso 5). Com isso, a quantidade de mensagens trocadas entre gerenciadores é praticamente constante. Em nossas simulações, observamos também que o tempo utilizado para a atualização do cache foi de 10 segundos, transferindo aproximadamente 5 KBytes no total.

5.2. Busca por Recursos Global

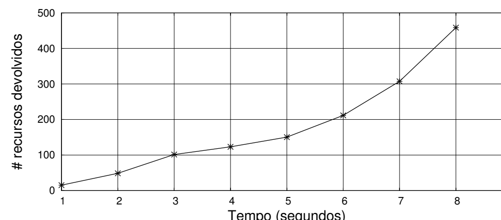
Nos experimentos realizados para buscas por recursos em domínios globalmente dispersos, o recurso utilizado foi a quantidade de memória RAM de uma máquina e o

intervalo de valores foi definido por todos os números inteiros positivos possíveis para esse recurso. Os parâmetros utilizados para a atualização da LBI foram: atualização do sucessor e do predecessor a cada 5 minutos, atualização da tabela de *fingers* e *repeated fingers* a cada 10 minutos.

A *quantidade de recursos visitados* por onde uma mensagem passa até encontrar o recurso procurado é muito importante para verificar a escalabilidade e o desempenho da solução proposta. Nesta simulação, adicionamos na LBI 1000 recursos com valores incrementais de um em um. Cada ponto da Figura 8(a) representa a quantidade de recursos visitados até encontrar o recurso procurado. Para encontrar o recurso com valor 300 é necessário visitar 14, 10 e 6 recursos com cinco, seis e sete *fingers* respectivamente. Esses valores são muito razoáveis, se comparados com a busca linear que visitaria 300 recursos e está de acordo com a análise teórica de que a quantidade de recursos visitados deve ser $O(\log n)$.



(a) Quantidade de recursos visitados em uma busca de acordo com a quantidade de $fingers$.



(b) Quantidade de recursos devolvidos dada uma busca por intervalo.

Figura 8. Quantidade de recursos visitados e devolvidos em uma busca global.

Um dos objetivos da LBI é estruturar os recursos de forma a poder localizar e devolver os recursos com valores compreendidos em um certo intervalo. Portanto realizamos experimentos para medir como a *quantidade de recursos devolvidos* em uma busca se comporta com o aumento do tempo da busca. Neste experimento adicionamos 1000 recursos, utilizamos uma tabela de *fingers* com 6 ponteiros e fizemos uma busca por todos os valores dos recursos que pertencem à LBI. Cada ponto da Figura 8(b) representa a quantidade de recursos devolvidos em um determinado momento. Por exemplo, se a busca dura 3 segundos, são devolvidos 100 recursos. Como o comportamento da busca, experimentalmente corresponde aproximadamente a uma função exponencial, o tempo total para obter todos os valores do intervalo será de $O(\log n)$, obtido da fórmula: se $quantidade_{devolvidos} = O(e^{tempo})$ então $tempo_{total} = O(\log(todos_{devolvidos}))$.

O *consumo de largura de banda utilizada em uma busca por intervalo* é mostrado na Figura 9. Cada ponto representa os KBytes por segundo necessários para obter todos os recursos inseridos na Grade. Neste experimento, adicionamos os recursos à LBI utilizando uma tabela de *fingers* com 6 ponteiros. Por exemplo, para obter 300 recursos, são necessários 20 KBytes/segundo. O comportamento da função corresponde aproximadamente a uma função linear, como mencionado na Seção 4.3.

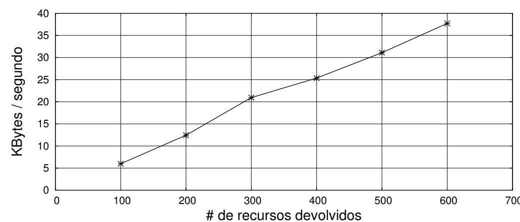


Figura 9. Consumo de largura de banda utilizada na busca por intervalo.

6. Trabalhos Relacionados

Uma das primeiras técnicas para descoberta de recursos em ambientes de Grade utilizando técnicas P2P foram descritas em [Iamnitchi et al. 2002, Iamnitchi and Foster 2001]. Essas técnicas, diferentemente da nossa, não contemplam a vizinhança em termos de latência. INS/TWINE [Balazinska et al. 2002], por sua vez, se concentra no armazenamento de dados (e.g., XML), mas não permite buscas por um intervalo de valores. Condor, inicialmente desenvolvido para o compartilhamento de recursos em redes locais, apresenta em [Butt et al. 2003] uma alternativa de utilizar as tabelas de roteamento [Rowstron and Druschel 2001] da DHT Pastry para localizar os gerenciadores. Nossa estratégia de busca na vizinhança, além de ser independente da DHT, utiliza a camada da rede que pode ajudar a encontrar gerenciadores mais próximos ainda. Globus, com seu serviço de monitoramento e descoberta [Zhang and Schopf. 2004] permite que um nó ("provedor de informação") possa obter recursos de um ou mais gerenciadores ("diretórios de agregado"). As diferenças com nossa estratégia são: (1) os nós devem conhecer os endereços dos diretórios aos quais se conectar e (2) não precisamos de servidores centrais, diretórios, para obter recursos da Grade. NodeWiz apresenta em [Basu et al. 2005] um modelo muito parecido ao do Globus, mas com melhorias nas buscas por intervalos. Da mesma forma que a nossa estratégia, a latência entre os gerenciadores é considerada importante. As diferenças com a nossa estratégia são: não temos diretórios considerados como servidores estáveis e o balanceamento de carga está distribuído em todos os gerenciadores da Grade – e não só nos servidores.

Em relação à estratégia de localização de recursos em domínios globalmente dispersos, uma das primeiras técnicas para resolver o problema das buscas por recursos por intervalo em uma Grade era usar uma extensão da DHT CAN [Ratnasamy et al. 2001]. Nesta proposta [Andrzejak and Xu 2002], cada gerenciador é adicionado à DHT e a mensagem de busca é propagada para todos eles. Com a nossa técnica, não temos a restrição de usar uma DHT específica e não usamos a propagação de mensagens, que pode levar a problemas de escalabilidade. *Prefix Hash Tree* (PHT) [Ramabhadran et al. 2004] é uma árvore de busca por intervalos, onde os valores dos recursos estão indexados nos nós intermediários e os recursos em si armazenados nas folhas. Nossa estratégia não utiliza uma estrutura extra, pois a ordem dos valores é dada pela própria LBI. No contexto da PHT, Gao et al. [Gao and Steenkiste 2004] utilizam os nós intermediários para armazenar os recursos. Para a busca dos recursos contidos em um intervalo, eles propõem duas alternativas: (1) ter um identificador para cada valor do intervalo e (2) ter só um identificador onde sejam armazenados todos os valores. O problema da (1) é que a busca tem que ser dividida nos valores do intervalo, enquanto o problema da (2) é que podem existir difer-

entes valores que compartilhem o mesmo identificador [Rao et al. 2003], e desta forma, o gerenciador responsável pelo único identificador poderá ter problemas de sobrecarga. *Skip Graph* [Aspnes and Shah 2003], que não utiliza a DHT para o armazenamento, é uma árvore de busca onde cada nível é uma lista ordenada pelos valores dos recursos. O último nível, muito parecido com a LBI, é formado por ponteiros que, no pior caso, podem resultar em uma busca linear. Nossa estratégia não utiliza uma estrutura adicional para ordenar os valores e a busca é realizada em tempo logarítmico.

7. Conclusões e Trabalhos Futuros

Neste artigo propomos duas novas estratégias para a busca de recursos em Grades Computacionais. A primeira lida com gerenciadores vizinhos em termos de latência, enquanto a segunda lida com domínios globalmente dispersos. Na busca por recursos na vizinhança utilizamos informações da camada de rede que ajudam a encontrar novos gerenciadores mais próximos. Na busca global propomos uma estrutura simples e eficiente baseada em uma lista distribuída e ordenada, para o registro e buscas dos recursos. É importante mencionar que quando um usuário realiza uma busca por recursos, as duas estratégias podem ser utilizadas de forma complementar. Por exemplo, se a quantidade de recursos desejada não é alcançada na vizinhança, o usuário pode também procurar por recursos utilizando a estratégia de busca global.

Além disso, mostramos que, de acordo com os resultados obtidos nas simulações, as estratégias propostas são assintoticamente tão eficientes quanto as propostas que usam propagação de mensagens e árvores distribuídas, com a vantagem de utilizar uma estrutura mais simples de administrar. Em relação à escalabilidade, observou-se que a quantidade de nós não influenciou no desempenho das estratégias propostas.

Atualmente estamos estudando alternativas para melhorar nossas estratégias. No caso da vizinhança, estamos analisando a retirada da DHT. Como conseqüência, cada gerenciador teria que ter armazenado localmente os roteadores da sua vizinhança. Desta forma, poderia haver um aumento na quantidade de mensagens trocadas para achar o gerenciador cujo roteador intercepte. No caso da busca global, estamos analisando alternativas para reduzir a largura de banda usada em uma busca. Para isso, o nó que passa a requisição de busca teria que enviar também os recursos que já foram visitados. Como conseqüência, evitar-se-ia que os recursos já visitados sejam novamente requisitados. Em respeito às simulações, para ambas alternativas, nosso próximo passo é incluir modelos de redes maiores e, finalmente, comparar nossas estratégias com as soluções similares.

O projeto InteGrade recebe apoio do CNPq (proc. 55.0094/2005-9) e Vladimir Rocha recebeu bolsa de mestrado da CAPES.

Referências

Adjie-Winoto, W., Schwartz, E., Balakrishnan, H., and Lilley, J. (1999). The Design and Implementation of an Intentional Naming System. In *Proc. of the 17th ACM Symposium on Operating Systems Principles*, pages 186–201.

Andrzejak, A. and Xu, Z. (2002). Scalable, Efficient Range Queries for Grid Information Services. In *Proc. of the Second IEEE International Conference on Peer-to-Peer Computing*.

Aspnes, J. and Shah, G. (2003). Skip Graphs. In *Proc. of the 14th Annual ACM SIAM Symposium on Discrete Algorithms*.

Balazinska, M., Balakrishnan, H., and Karger, D. (2002). INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery. In *Proc. of the First International Conference on Pervasive Computing*, pages 195–210, London, UK. Springer-Verlag.

Basu, S., Banerjee, S., Sharma, P., and Lee, S.-J. (2005). NodeWiz: Peer-to-peer Resource Discovery for Grids. In *Proc. of the 5th International Workshop on Global and Peer-to-Peer Computing in conjunction with CCGrid*.

Butt, A. R., Zhang, R., and Hu, Y. C. (2003). A Self-Organizing Flock of Condors. In *Proc. of the 15th ACM/IEEE Conference on Supercomputing*. IEEE Computer Society.

Cohen, E. and Shenker, S. (2002). Replication strategies in unstructured peer-to-peer networks. In *Proc. of the ACM SIGCOMM conference on applications, technologies, architectures, and protocols for computer communications*, pages 177 – 190, Pittsburgh, Pennsylvania, USA.

Cosway, P. R. (1997a). Replication control in distributed B-trees. Technical Report MIT/LCS/TR-705.

Cosway, P. R. (1997b). Replication Control in Distributed B-Trees. Technical report, Cambridge, MA, USA.

de Camargo, R. Y., Goldchleger, A., Carneiro, M., and Kon, F. (2004). Grid: An Architectural Pattern. In *Proc. of the 11th Conference on Pattern Languages of Programs*, Monticello, Illinois.

Fitzgerald, S. (2001). Grid information services for distributed resource sharing. In *Proc. of the 10th IEEE International Symposium on High Performance Distributed Computing*, page 181, Washington, DC, USA. IEEE Computer Society.

Fitzgerald, S., Foster, I., Kesselman, C., von Laszewski, G., Smith, W., and Tuecke, S. (1997). A Directory Service for Configuring High-performance Distributed Computations. In *Proc. 6th IEEE Symp. on High Performance Distributed Computing*, pages 365–375. IEEE Computer Society Press.

Foster, I. and Kesselman, C. (2003). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann Publishers Inc., second edition.

Gao, J. and Steenkiste, P. (2004). An Adaptive Protocol for Efficient Support of Range Queries in DHT-Based Systems. In *Proc. of the 12th IEEE International Conference on Network Protocols*, pages 239–250.

Gummadi, K., Gummadi, R., Gribble, S., Ratnasamy, S., Shenker, S., and Stoica, I. (2003). The impact of DHT routing geometry on resilience and proximity. In *Proc. of the ACM SIGCOMM conference on applications, technologies, architectures, and protocols for computer communications*, pages 381–394, Karlsruhe, Germany.

Gummadi, K. P., Saroiu, S., and Gribble, S. D. (2002). King: Estimating Latency Between Arbitrary Internet End Hosts. In *Proc. of the Second ACM SIGCOMM Workshop on Internet measurment*, pages 5–18, New York, NY, USA. ACM Press.

Iamnitchi, A., Foster, I., and Nurmi, D. C. (2002). A Peer-to-Peer Approach to Resource Location in Grid Environments. In *Proc. of the 11th IEEE International Symposium on High Performance Distributed Computing*, page 419, Washington, DC, USA. IEEE Computer Society.

Iamnitchi, A. and Foster, I. T. (2001). On Fully Decentralized Resource Discovery in Grid Environments. In *Proc. of the Second International Workshop on Grid Computing*, pages 51–62, London, UK. Springer-Verlag.

Leuf, B. (2002). *Peer to Peer*. Addison-Wesley.

Lv, Q., Cao, P., Cohen, E., Li, K., and Shenker, S. (2002). Search and Replication in Unstructured Peer-to-Peer Networks. In *Proc. of the ACM SIGMETRICS international conference on measurement and modeling of computer systems*, pages 258–259.

Nelson Minar et al. (2001). *Peer-to-Peer – Harnessing the Power of Disruptive Technologies*. O'Reilly.

Ramabhadran, S., Ratnasamy, S., Hellerstein, J. M., and Shenker, S. (2004). Brief announcement: Prefix Hash Tree. In *Proc. of the 23rd ACM Symposium on Principles of Distributed Computing*, page 368.

Rao, A., Lakshminarayanan, K., Surana, S., Karp, R., and Stoica, I. (2003). Load Balancing in Structured P2P Systems. In *Proc. of the Second International Workshop on Peer-to-Peer Systems.*, pages 68–79.

Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Schenker, S. (2001). A Scalable Content-Addressable Network. In *Proc. of the ACM SIGCOMM conference on applications, technologies, architectures, and protocols for computer communications*, pages 161–172.

Rhea, S., Geels, D., Roscoe, T., and Kubiatowicz, J. (2004). Handling Churn in a DHT. In *Proc. of the 2004 USENIX Annual Technical Conference*, Boston, Massachusetts.

Rocha, V. M. (2005). Protocolos P2P para Interligação de Aglomerados em Grades Computacionais. Master's thesis, Instituto de Matemática e Estatística, Universidade de São Paulo.

Rowstron, A. and Druschel, P. (2001). Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proc. of the IFIP/ACM International Conference on Distributed Systems Platforms*, volume 2218 of *Lecture Notes in Computer Science*, pages 329–350, Heidelberg, Germany. Springer.

Stallings, W. (2003). *Data and Computer Communication*. Maxwell Macmillan International, seventh edition.

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 *Proc. of the ACM SIGCOMM conference on applications, technologies, architectures, and protocols for computer communication*, pages 149–160.

Tanenbaum, A. (2002). *Computer Networks*. Prentice Hall.

Zhang, X. and Schopf., J. (2004). Performance Analysis of the Globus Toolkit Monitoring and Discovery Service, MDS2. In *Proc. of the International Workshop on Middleware Performance*.