Protocolos par-a-par para interligação de aglomerados em grades computacionais

Vladimir Moreira Rocha

Dissertação apresentada ao
Instituto de Matemática e Estatística
da Universidade de São Paulo
para obtenção do grau de
Mestre em Ciências da Computação

Área de Concentração: **Ciência da Computação**
Orientador: **Prof. Dr. Fabio Kon**

São Paulo, dezembro de 2005

Durante a elaboração deste trabalho o autor recebeu apoio financeiro da CAPES.

Protocolos Par-a-Par para Interligação de Aglomerados em Grades Computacionais

Este exemplar corresponde à redação
final da dissertação devidamente
corrigida e defendida por
Vladimir Moreira Rocha
e aprovada pela comissão julgadora.

São Paulo, dezembro 2005.

Banca examinadora:

- Prof. Dr. Fabio Kon (Orientador) - IME-USP
- Prof. Dr. Marcelo Finger - IME-USP
- Prof. Dr. Renato Cerqueira - PUC-Rio

Agradecimentos

Vou começar por Deus, agradecer Ele por ter me dado a vida e por estar aqui. Ainda não sei muito bem qual o destino que Ele me deu, mas um dia vou concluí-lo. Aos meus pais, por seu apoio incondicional, pelos momentos vividos, pelo carinho, amor e por acreditar sempre em mi.

Agradeço profundamente ao professor Fabio Kon pelas suas orientações sempre acertadas bem como pela sua amizade.

Agora aos amigos, quero agradecer aos meus "irmãos" da Republica e da vida, Edu, Thiaguin, Christian, Gordito e Karina. Com eles os momentos vividos e que vamos viver estarão sempre comigo, são inesquecíveis. Yrla, você merece um beijo especial.

Gostaria de agradecer também aos professores que participaram tanto na banca da defesa como da qualificação, Marcelo Finger, Renato Cerqueira e Alfredo Goldman.

É claro que não vou esquecer dos membros do GSD e do InteGrade, principalmente pelas reuniões, pelas comidas e pelos temas engraçados que as vezes abordávamos.

Finalmente, agradeço ao IME e à USP por ter me dado a oportunidade de continuar meus estudos de pós-graduação e à CAPES pelo apoio financeiro.

Vladimir Moreira Rocha

Universidade de São Paulo
Dezembro 2005

Resumo

Neste trabalho são apresentados dois novos protocolos *Par-a-Par* que servirão para interligar aglomerados em uma Grade Computacional e localizar, nos gerenciadores desses aglomerados, os recursos oferecidos por eles.

O primeiro protocolo permite que os aglomerados de uma Grade Computacional se conheçam e que a latência entre eles seja a menor possível. Para isto utiliza-se a função de roteamento da camada de rede para descobrir novos aglomerados que estão dentro de uma rota entre dois aglomerados.

O segundo protocolo permite localizar informações nos diferentes aglomerados. A localização destas informações pode ser feita por intervalo de valores, e para isto é criada uma estrutura de lista distribuída ordenada pelos valores das informações.

Os protocolos foram simulados e testados em relação a vários pontos críticos dos protocolos *Par-a-Par*, como escalabilidade, tempo de ingresso e largura de banda utilizada. Mostramos que os resultados da simulação destas novas propostas cumprem com os objetivos de escalabilidade e eficiência.

Abstract

In this work, we present two new Peer-to-Peer protocols that allow to interconnect Grid clusters and to locate, in the clusters manager, shared resources.

The first protocol allows to link different Grid clusters and to organize them to obtain the less latency possible among them. To accomplish these objetives, we utilize the routing function present in the network layer. With that, it is possible to find new clusters present in a path between two clusters.

The second protocol allows to locate information shared by the Grid clusters. The location of these information could be made by a value range. In order to perform a location, a distributed list structure is created, ordered by the information values.

Finally, we simulate these protocols in many critical points of Peer-to-Peer protocols like, scalability, join time and bandwidth used. We show the results and prove that they satisfy scalability and efficiency goals.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Seja no âmbito comercial, industrial ou de pesquisa, faz-se necessário apresentar e distribuir alguns serviços para as pessoas. Com a chegada da Internet, passou a ser possível acessar estes serviços de qualquer parte do mundo através de um computador. Só é preciso saber como encontrar as máquinas que estão conectadas à rede e que disponibilizam tais serviços.

A arquitetura computacional mais utilizada que permite expor esses serviços é a cliente/servidor. Nesta arquitetura, cada computador cumpre um papel bem definido. Se o computador é um servidor, então possui certas características, como por exemplo, ser um computador poderoso em termos de velocidade de processamento, armazenamento e administração dos pedidos dos clientes para evitar congestionamentos e conseguir oferecer os serviços com qualidade. Se for um cliente, ele pode acessar os serviços disponíveis pelo servidor para seu aproveitamento.

Como exemplo, desde o começo dos anos 90, existiam servidores que ofereciam arquivos de música e outros documentos para compartilhamento e eram acessados por um número pequeno de clientes. Quando a Internet foi se massificando, surgiram problemas de como administrar o crescimento exponencial do número de clientes que requisitavam estes arquivos e como distribuir e localizar as informações disponibilizadas pelos milhares de novos servidores que apareceram.

Para atender estes problemas foram desenvolvidos dois ambientes, que surgiram de diferentes comunidades e portanto foram projetados para satisfazer diferentes necessidades: as redes *Par-a-Par* e os sistemas de Computação em Grade.

As redes *Par-a-Par* surgiram para atender a necessidade de compartilhar arquivos entre milhares de computadores sem precisar dos servidores.

Cada um destes computadores possui as mesmas responsabilidades e capacidades, podendo ser ao mesmo tempo servidor e cliente. Isto era uma proposta muito diferente da arquitetura cliente/servidor que se tinha até o momento [eA01]. Neste tipo de rede, a comunicação é feita diretamente entre computadores e não por intermédio de servidores. Estes computadores são chamados pares. Assim, se um cliente precisa de um serviço, simplesmente busca outros pares que o ofereçam.

Dentro dos protocolos de comunicação *Par-a-Par*, existem diversos aspectos que foram abordados nos últimos anos. No entanto, os pesquisadores têm se concentrado na construção de sistemas e arquiteturas que descobrem e localizam recursos que existem em um outro computador, tais como documentos e arquivos de música, e na escalabilidade do sistema para milhares de pares conectados.

No caso dos sistemas de Computação em Grade, eles surgiram nos anos 80 para atender a necessidade do compartilhamento de recursos para o uso intensivo da computação, para processamento de dados, simulações, etc. Para aproveitar e melhorar esta nova idéia, os pesquisadores da área desenvolveram serviços sofisticados que conectam vários computadores para prover uma maior capacidade de processamento na execução de tarefas.

A localização dos recursos neste tipo de sistema não apresenta problemas quando são poucos os computadores envolvidos. Mas, com o crescimento de sistemas de grade computacionais, passou a ser possível a computação distribuída entre aglomerados de computadores geograficamente distantes. Assim não só é possível executar uma tarefa em um outro aglomerado, como também acessar os seus recursos.

1.1 Motivação

O InteGrade [GKG+04] [Int] é um sistema de Computação em Grade que interliga computadores com a finalidade de usar a capacidade ociosa de cada um deles em tarefas de computação de alto desempenho.

Sua arquitetura basicamente se resume a máquinas que fazem parte de um aglomerado e um nó gerenciador responsável por administrá-los. Os módulos que coletam as informações disponíveis sobre um nó, como memória RAM, Disco Rígido e Sistema Operacional, são chamados LRM (*Local Resource Manager*). O módulo gerenciador, chamado GRM (*Global Resource Manager*) coleta as informações dos LRM.

No atual estágio do sistema, as aplicações são executadas em um só aglomerado. Se a tarefa não pode ser executada no aglomerado local, a solicitação de execução da tarefa tem que ser enviada posteriormente até poder ser realizada.

A motivação do presente trabalho é criar um novo protocolo para redes *Par-a-Par* que servirá para interligar aglomerados no InteGrade. Com esses protocolos será possível fazer com que os GRMs de diferentes aglomerados possam se conhecer e localizar, nos GRMs da rede, informações que são modificadas periodicamente, as quais chamaremos de *informações dinâmicas*.

No caso da interligação de GRMs, o objetivo é que a comunicação entre eles tenha baixa latência. Assim, se uma tarefa não pode ser executada num aglomerado, poderá ser transferida para outro com uma perda de desempenho relativamente menor. No caso da localização, as informações podem ser recursos disponíveis coletadas pelos LRMs. Com isso um usuário do sistema poderá encontrar, por exemplo, 64 máquinas (LRM) que tenham como mínimo 256 MB de RAM disponível, no momento, para executar uma multiplicação de matrizes.

1.2 Objetivos

Os principais objetivos do presente trabalho são:

- Criar um protocolo que permita estruturar os pares de uma rede de forma que a comunicação entre eles seja a mais rápida possível. (Protocolo de Interligação).
- Criar um protocolo que permita localizar os recursos disponibilizados pelos pares da rede. As buscas por estes recursos poderão ser definidas para um intervalo dos valores desses recursos. (Protocolo de Localização).

Os objetivos específicos traçados são:

- Os protocolos desenvolvidos para redes *Par-a-Par* poderão se adequar aos sistemas de Computação em Grade e vice-versa. Por tanto, esses protocolos deveriam ter as seguintes características: independência de um administrador central, suporte para buscas baseadas em atributos e escalabilidade [IFN02].
- Utilizar uma estrutura *Par-a-Par* que permita a busca de um recurso em tempo sub-linear. Esta estrutura será a tabela de *hash* distribuída.

- Avaliar a escalabilidade dos protocolos desenvolvidos, utilizando um simulador de redes de grande área.

1.3 Contribuições

As contribuições deste trabalho correspondem aos protocolos criados para cumprir os objetivos traçados na sua concepção.

No caso do Protocolo de Interligação, é inovador a utilização da função de roteamento da camada de rede para que um par possa encontrar novos pares que possam estar mais perto em termos de latência.

Em relação ao Protocolo de Localização, foi desenvolvida uma nova estrutura que permite uma busca pelo intervalo dos valores dos recursos. Esta estrutura, baseada na tabela de *hash* distribuída, corresponde a uma lista distribuída, ordenada pelos valores dos recursos e que é mais simples de administrar que as estruturas baseadas em árvores.

1.4 Estrutura da Dissertação

Esta dissertação está organizada como a seguir. No Capítulo 2 apresentamos a arquitetura do InteGrade que utilizará os protocolos para localização e comunicação entre aglomerados. O Capítulo 3 é dedicado à descrição das redes *Par-a-Par*. No Capítulo 4, descrevemos como funciona a estrutura que será a base do protocolo e que permite uma localização em tempo $O(\log(n))$ de informações espalhadas pela rede. No Capítulo 5, descrevemos o Protocolo de Interligação que propõe resolver o problema de como obter uma comunicação eficiente entre os pares. No Capítulo 7 apresentamos alguns dos sistemas existentes relacionados ao Protocolo de Interligação. O Capítulo 6 descreve o Protocolo de Localização, que servirá para buscar nos GRMs informações dinâmicas dos LRM. Alguns dos sistemas existentes relacionados ao Protocolo de Localização são apresentados no Capitulo 7.2. O Capítulo 8 mostra os resultados da simulação dos protocolos apresentados, em redes de grande área. Finalmente, as conclusões e trabalhos futuros a serem desenvolvidos como continuação deste trabalho são apresentadas no Capítulo 10.

Capítulo 2

InteGrade

O InteGrade [Int] é um sistema de computação em grade desenvolvido por várias universidades brasileiras: IME/USP, DCT/UFMS, DI/PUC-Rio, DEINF/UFMA e que tem por objetivo o compartilhamento de computadores pessoais de maneira a permitir a utilização de sua capacidade ociosa em tarefas de computação de alto desempenho, sem nenhum custo adicional em termos de hardware, nem exigindo nenhuma mudança drástica na instalação de software nas máquinas.

O InteGrade está sendo construído usando uma implementação de uma arquitetura padronizada, CORBA, que permite a objetos distribuídos se comunicarem [Gro02]. Na arquitetura CORBA são descritos serviços, como Serviço de Nomes, Serviço de Negociação e Serviço de Transações os quais já estão implementados, testados e que ajudam a diminuir a complexidade do sistema e custos de manutenção do software, permitindo que o desenvolvimento do projeto somente fique concentrado na domínio da aplicação.

Segue abaixo um resumo dos diferentes aspectos dentro do InteGrade que formam a arquitetura interna do sistema e que permitem o seu funcionamento.

- **Detecção e Análise de Padrões de Uso**

  O modulo de Análise e Detecção de Padrões de Uso é o encarregado por coletar longas séries de informações referentes ao uso de recursos de cada uma das máquinas da Grade. Ele fornecerá, ao escalonador de aplicações, informações que melhorarão a sua capacidade de decisão de onde alocar a tarefa a executar. Por exemplo, se uma determinada máquina pertencente à Grade é usada de maneira intermitente ao longo do dia, estando ociosa apenas em breves períodos, o escalonador analisará que

é pouco eficiente escalonar uma tarefa que precisa de um longo período de execução, resultando no cancelamento ou migração da tarefa. Entretanto, convém lembrar que esse módulo somente fornecerá dicas, ou seja, a chance do padrão sugerido ocorrer tende a ser grande, mas de maneira alguma representa uma certeza.

- **Segurança**

  Um dos problemas sérios que temos em aplicações e sistemas que usam as redes é a segurança. Esse módulo pretende assegurar que as máquinas da grade sejam protegidas contra ataques, preservando assim a integridade dos dados da máquina do provedor de recursos. Com isso, evita-se que um usuário possa tentar roubar dados sigilosos e arquivos pessoais, prejudicar ou impedir o correto funcionamento da máquina e também instalar programas que causem perdas de arquivos e dados. Assim, são necessários medidas que impeçam que tal tipo de ataque ocorra.

- **Suporte a aplicações paralelas**

  O suporte de aplicações paralelas consiste em permitir o uso de aplicações paralelas que precisam de uso simultâneo de vários recursos e que geralmente são executadas em sistemas de computação em grade. No momento, poucos sistemas de grade oferecem suporte à execução de aplicações paralelas fortemente acopladas, especialmente quando não há nenhuma garantia sobre a disponibilidade dos recursos. InteGrade oferece suporte ao modelo BSP (*Bulk Synchronous Parallelism*) [Val90] com o objetivo de que as aplicações já existentes possam rodar na Grade. Assim, junto com o módulo de Análise e Detecção de Padrões de Uso, o InteGrade poderá oferecer uma previsão muito provável sobre a disponibilidade de recursos em cada máquina do sistema.

- ***Checkpointing***

  O mecanismo de checkpointing provê recuperação por retrocesso para aplicações sendo executadas na Grade. Isto é muito importante devido ao fato de máquinas poderem ficar inacessíveis ou mudar seu estado de ocioso a ocupado rapidamente, comprometendo a execução das aplicações nessas máquinas. O mecanismo consiste em armazenar periodicamente o estado de um processo, de modo que este estado possa ser recuperado em caso de falha em sua execução.

A arquitetura do InteGrade é composta por aglomerados interligados. Cada aglomerado contém normalmente entre duas e 100 máquinas. Vamos descrever a arquitetura de um aglomerado típico e a arquitetura entre aglomerados.

2.1 Arquitetura Intra-Aglomerado

A Figura 2.1 apresenta os elementos mais importantes de um aglomerado do InteGrade. Existem vários tipos de nós no aglomerado. O Nó Dedicado é uma máquina reservada à computação em grade, assim como se fosse um nó em um aglomerado dedicado tradicional. Tais máquinas não são o foco principal do InteGrade, mas tais recursos podem ser integrados aos aglomerados se desejado. O Nó Compartilhado é aquele que pertence a um usuário que compartilha seus recursos ociosos com a grade. Finalmente, o Nó de Usuário é aquele que possui a capacidade de submeter aplicações para serem executadas na grade. O Gerenciador de Aglomerado é responsável pela execução de módulos responsáveis pela coleta de informações e escalonamento. É importante destacar que um nó pode pertencer a várias categorias simultaneamente, por exemplo, um nó que tanto compartilha seus recursos ociosos quanto é capaz de submeter aplicações para serem executadas na grade.

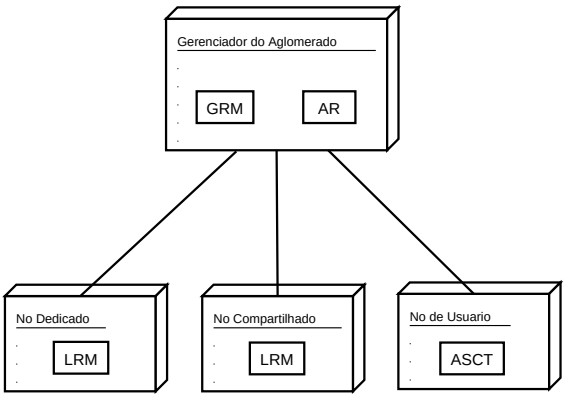


Figura 2.1: Arquitetura Intra-Aglomerado do InteGrade.

Os módulos apresentados na Figura 2.1 são responsáveis pela execução de diversas tarefas necessárias à Grade, daremos a seguir uma descrição deles.

O ASCT (*Application Submission and Control Tool*) é uma ferramenta que permite aos usuários da Grade submeter aplicações que serão executadas

na mesma. Com essa ferramenta é possível estabelecer certas condições de execução, como: sistema operacional, configurações de hardware e software, etc. e preferências, como recursos necessários (por exemplo quantidade de memória RAM mínima) para melhorar o desempenho da execução da aplicação. Essa ferramenta também disponibiliza ao usuários o controle, monitoramento e recuperação dos resultados da execução da tarefa. O LRM (*Local Resource Manager*) é executado em todas as máquinas que compartilham seus recursos com a Grade e é responsável pela coleta de informações sobre a disponibilidade de recursos em um dado nó. Também é responsável por exportar os recursos desse nó à grade, permitindo à execução de aplicações submetidas por usuários da grade. O GRM (*Global Resource Manager*) geralmente é executado em um nó que gerencia o aglomerado e é responsável por coletar as informações dos LRM, assim como tomar decisões de escalonamento baseadas em tais informações [GKG+04]. O AR (*Application Repository*) é o responsável por armazenar as aplicações submetidas pelos usuários e que serão executadas na Grade. Esse repositório de aplicações fornece também outras funcionalidades como por exemplo o registro de meta-dados da aplicação, como nome, tipo de aplicação e sistema operacional onde pode ser executado. Finalmente, existem outros módulos como o LUPA, GUPA e o NCC que saem do escopo deste trabalho, mas que são analisados em [Gol04].

A colaboração entre esses módulos é dada pelos protocolos responsáveis pela disseminação de informações e pela execução de tarefas e que são derivados dos protocolos utilizados pelo sistema operacional distribuído 2K [KCM+00].

2.1.1 Protocolo de Disseminação de Informações

Este protocolo permite a um aglomerado do InteGrade manter atualizadas as informações dos recursos disponibilizados nas diversas máquinas do aglomerado. Essas informações podem ser dados estáticos (arquitetura da máquina, sistema operacional, memória total disco rígido, etc.) ou dados dinâmicos (porcentagem de CPU ociosa, memória RAM disponível no momento, etc.) e que permite que o GRM escalone as tarefas da forma mais adequada aos requisitos da aplicação submetida.

Para manter os dados do aglomerado atualizados, periodicamente cada LRM verifica a disponibilidade dos seus recursos. Caso exista alguma mudança significativa entre a informação atual e a informação anterior, o LRM envia ao GRM essa informação para que seja atualizada. Para manter a informação das máquinas do aglomerado que estão no ar, cada LRM envia periodicamente

ao GRM uma mensagem de vida (*keep-alive*).

2.1.2 Protocolo de Execução de Aplicações

Este protocolo têm por objetivo a execução de uma aplicação submetida sobre recursos compartilhados da Grade. A Figura 2.2 apresenta os passos necessários para executar uma aplicação. Depois que a aplicação foi submetida ao repositório de aplicações através do ASCT, o usuário pede a este a execução de uma tarefa o qual é encaminhada ao GRM (1). O GRM, com as informações coletadas da Grade, procura os nós candidatos que cumprem com os requisitos da tarefa (2). Caso existam nós disponíveis, o GRM verifica se eles realmente possuem os recursos necessários para a execução da tarefa e pede para executá-la nesses nós (3). Devemos lembrar que o GRM somente tem uma visão aproximada da Grade. O LRM que aceita a execução da tarefa pede para o Repositório de Aplicações o arquivo executável da aplicação (4) e para o nó requisitante os arquivos de entrada (5). Finalmente executa a aplicação (6).

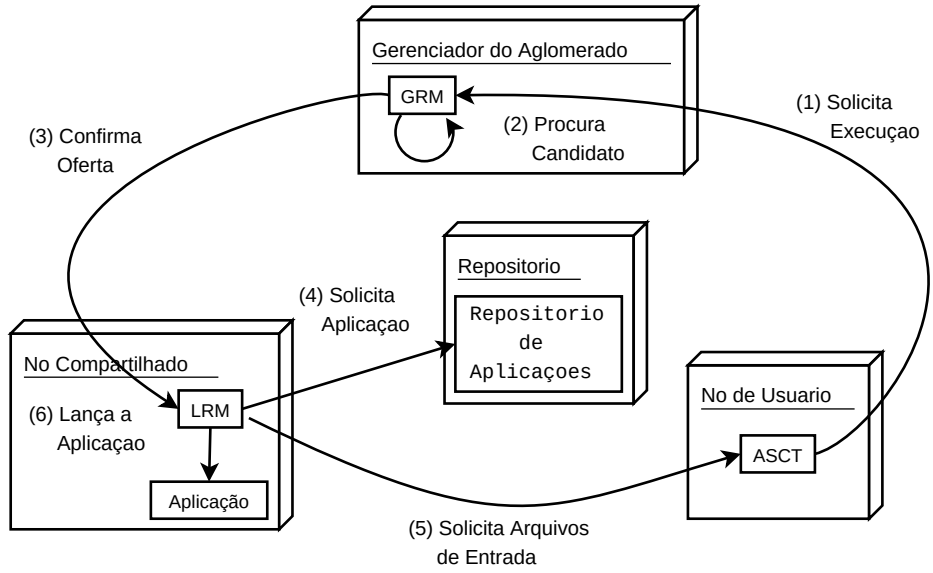


Figura 2.2: Protocolo de Execução de Aplicações.

2.2 Arquitetura Inter-Aglomerados

Uma primeira proposta para interligar eficientemente diversos aglomerados do InteGrade foi a de integrar os diferentes aglomerados em uma hierarquia de árvore definida pelos administradores dos aglomerados, como mostra a Figura 2.3. Um dos problemas que esta alternativa apresenta é que a escalabilidade do sistema pode ser comprometida de duas formas: a primeira é que os administradores tem que conhecer os endereços dos gerenciadores do aglomerado para conectá-los manualmente e no caso de haver muitos gerenciadores pode ficar difícil de administrar. A segunda é que um nó mantém informações sobre a sub-árvore de sua hierarquia e, no caso dessa sub árvore ser muito grande, a atualização das informações nesse nó pode gerar uma carga muito alta. Mesmo assim, no caso de se ter centenas de aglomerados do InteGrade esta proposta parece ser muito eficiente.

Como um dos objetivos do InteGrade é implementar um sistema capaz de atender potencialmente milhares de máquinas conectadas através de uma rede de grande área como a Internet, é necessário estender a arquitetura para interligar vários aglomerados. O objetivo deste projeto é oferecer o máximo de escalabilidade e velocidade de comunicação entre eles.

A arquitetura Inter-Aglomerados, terá dois protocolos fundamentais para seu funcionamento: O Protocolo de Interligação e o Protocolo de Localização de Recursos. O Protocolo de Interligação, como veremos na Seção 5, conecta os Aglomerados de uma maneira eficiente. O Protocolo de Localização procura por nós dos aglomerados que satisfaçam certos critérios, descritos na Seção 6.

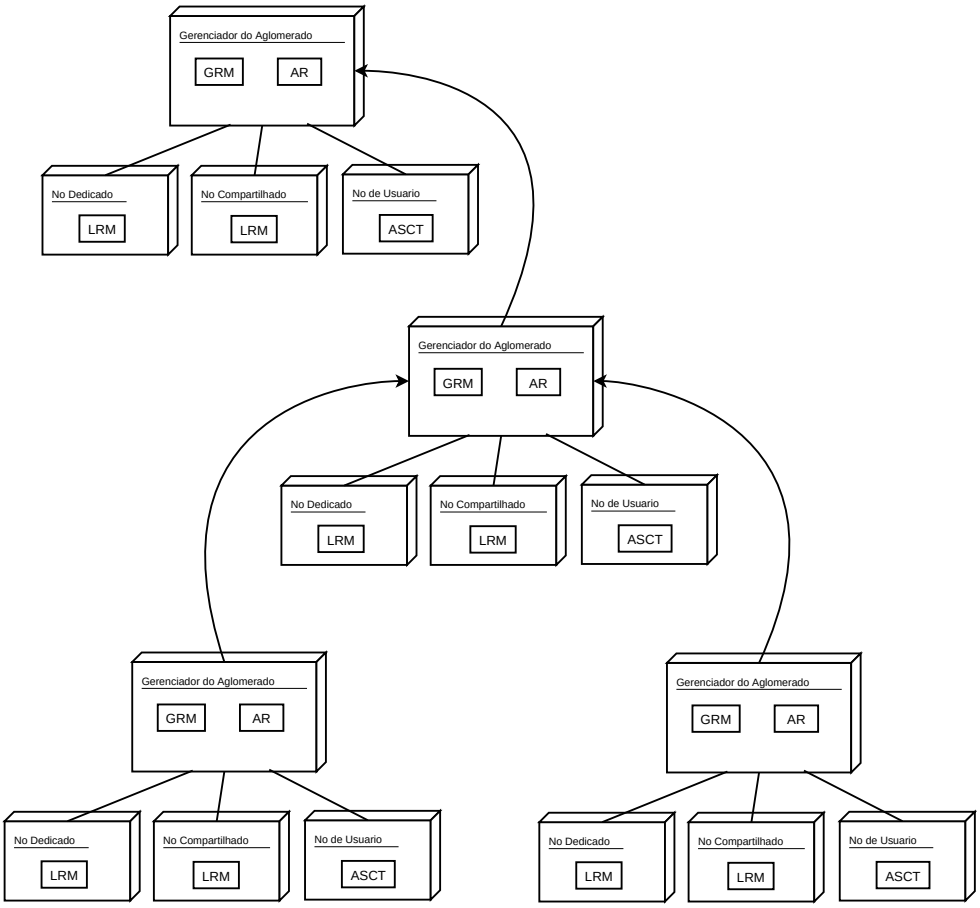


Figura 2.3: Hierarquia de aglomerados.

Capítulo 3

Redes P2P

O termo *Par-a-Par* (P2P ou *Peer-to-Peer*) como conceito foi usado pela primeira vez por vendedores de redes locais em meados dos anos 80 para descrever a conectividade de uma arquitetura de uma rede local [Leu02].

Mesmo que algumas aplicações *Par-a-Par* existam desde os finais dos anos 70, como a Usenet, que permitia o compartilhamento de notícias nas comunidades UNIX, foi só no ano 2000, com o surgimento de uma aplicação que compartilha arquivos de música chamada **Napster**, que o conceito começou a se generalizar e ser conhecido [STDG01]. Desde então, em meios não acadêmicos, *Par-a-Par* virou quase um sinônimo de aplicações para compartilhamento de arquivos.

Daremos a seguir uma definição do que é uma rede *Par-a-Par*. Como conceito, estará dividida em vários aspectos que juntos formarão uma definição compreensível.

- *Comunicação direta entre os pares*: A comunicação entre os pares, que podem ser pessoas, máquinas ou programas, não possui um mediador central que a controle. É possível ter uma comunicação entre dois pares quaisquer pertencentes à rede *Par-a-Par*.

- *A conectividade é usualmente transitória e não permanente*: Cada par pertencente à rede *Par-a-Par* tem uma período de vida curto (normalmente poucas horas [GDS+03, SGG02, CLL02]). Isso implica que nesta topologia, os serviços oferecidos e a comunicação são variáveis e intermitentes.

- *Está focado principalmente em pessoas e recursos*: As redes *Par-a-Par* proporcionam os meios necessários para obter qualquer tipo de recurso

que é oferecido pelos seus pares. Cada par tem um identificador global que o caracteriza como único.

- *Alternativa às estruturas baseadas em servidores*: A maioria das tecnologias criadas para as redes *Par-a-Par* tem como objetivo evitar as arquiteturas onde o par só atua como cliente ou só como servidor.

3.1 Arquiteturas

Existem vários modelos arquiteturais para redes *Par-a-Par* [Min01]. A seguir serão mostradas as características, vantagens, desvantagens, ingressos e buscas de quatro desses modelos que consideramos como os mais usados e conhecidos.

3.1.1 Modelo Atômico

O modelo atômico é chamado pelos puristas de "a verdadeira rede *Par-a-Par*". Este modelo apresenta algumas características que diferem totalmente das arquitetura baseadas em servidores. Primeiro, não existem servidores centrais que possam ter informação completa da rede e dos seus usuários, de modo que todos os pares são por sua vez clientes e servidores. Segundo, a estrutura de conexão formada por eles é aleatória, uma vez que um par pode se conectar potencialmente a qualquer outro par da rede. Terceiro, cada par é autônomo e administra seus próprios recursos e suas próprias conexões. Nesse caso, cada um deles mantém contato direto (conexão direta) com alguns outros, mas a comunicação pode ser estabelecida com qualquer outro par da rede (conexão virtual), como mostra a Figura 3.1.

Nos últimos tempos, algumas pesquisas em redes *Par-a-Par* têm utilizado uma variação do modelo atômico. Essa variação consta da criação de uma nova camada de pares, conhecidos como *super pares*, que tem como característica uma melhor capacidade de processamento, armazenamento, conectividade e confiabilidade. Com essa variação se tenta criar uma estrutura de pares que permita melhorar o desempenho nas buscas.

Escalabilidade é uma questão importante nos modelos arquiteturais das redes *Par-a-Par* onde a quantidade de pares que a utilizam é muito grande. Algumas pesquisas [Rit01] mostram que aplicações que utilizam este modelo, como Gnutella 1, são escaláveis no máximo a alguns milhares de pares conectados. Passado esse valor, o desempenho nas buscas, devido à sobrecarga na

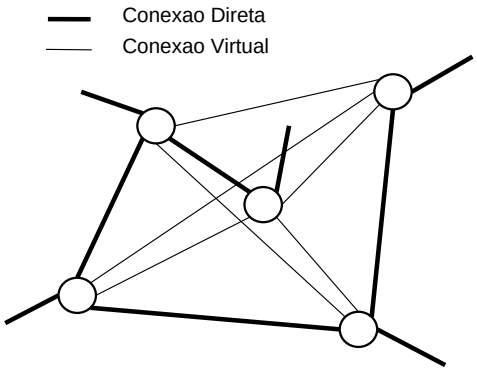


Figura 3.1: Modelo Atômico.

rede, começa a degradar exponencialmente até a rede ficar indisponível. Existem dois tipos de alternativas ao problema mencionado acima. A primeira é estruturar os pares como veremos na Seção 3.1.4 e a segunda é utilizar uma propagação controlada na comunicação entre pares, mostrado em "Buscas na rede" deste modelo.

Ingresso na rede. No modelo atômico, como não existe um servidor central de onde poder-se-ia obter outros pares com os quais se comunicar, existe um problema fundamental de como se unir à rede.

Existem dois métodos tradicionais para resolver em parte esse problema, cada um deles usado para diferentes propósitos. A primeira é propagar uma mensagem de pedido de ingresso usando *broadcast* para obter respostas de pares que estejam escutando esse tipo de mensagens. Essa proposta é muito utilizada e eficiente para redes locais, onde determinar os pares que estão escutando uma requisição de ingresso não é difícil. A segunda alternativa é se conectar a pares cujos endereços são conhecidos, geralmente utilizada nos casos onde os pares estão espalhados em uma rede grande, por exemplo uma WAN (*Wide Area Network*). A propagação de uma mensagem de ingresso explicada na primeira alternativa pode resultar em um grande consumo de largura de banda, pois a mensagem será propagada pela rede para todos os pares sem considerar se eles estão, ou não, escutando pela requisição.

Buscas na rede A base de todas as buscas por recursos neste modelo é a propagação da mensagem de busca nos pares da rede. Uma das primeiras

alternativas utilizadas nos protocolos de busca foi propagar a requisição a "todos" os pares da rede, chamado de *flooding*. A desvantagem desta alternativa se produz quando a quantidade de pares que utilizam a rede aumenta. Neste caso, a quantidade de mensagens de busca aumenta de forma exponencial e a rede fica inutilizada rapidamente pela sobrecarga. Surgiu então uma variação desta alternativa que utiliza uma variável TTL (*Time To Live*) que define o tempo de vida da mensagem, ou seja limita a propagação. O valor desta variável diminui em cada propagação da mensagem, o que controla a quantidade de mensagens que utilizam a rede. Com esta proposta, apesar da melhora do problema da sobrecarga da rede, não podemos ter certeza se o recurso existe ou não. Como a propagação da mensagem é controlada, ela somente alcança um certo nível de pares que podem não conter o recurso procurado e que pode estar além da barreira definida pela TTL. Outras alternativas mais recentes, como a do protocolo do Gnutella 2 [gnua], propõem que um par (conhecido como *super par ou hub peer*) armazene os nomes dos recursos dos seus conhecidos. Com isso, quando a esse par chega uma mensagem de busca por um recurso x, este encaminha diretamente ao par que possui x, o que evita, de certa forma, a propagação da mensagem e melhora o desempenho de algumas buscas.

3.1.2 Modelo Centrado no Usuário

O modelo centrado no usuário, diferentemente do modelo atômico, é gerenciado por um servidor central que atua como mediador para ingresso e buscas de pares da rede, como mostra a Figura 3.2. Esse servidor contém um diretório, que mantém os endereços (chamado de *links* persistentes) dos pares conectados na rede e que possibilita a procura por eles. É atualizado enviando periodicamente uma mensagem de vida (*heartbeat*) aos pares conectados. Essa arquitetura é uma das mais utilizadas devido à facilidade de conhecer os outros pares conectados à rede e por ser bem parecida com a arquitetura Cliente/Servidor.

A escalabilidade deste tipo de modelo é dada pela sobrecarga exercida no servidor central. Para saber se o modelo pode ser considerado escalável, ou não, para uma quantidade grande de pares conectados, devemos analisar qual é a função do administrador central. Se a função dele for somente a de permitir o ingresso de pares à rede, então todas as transferências de mensagens (como as de busca) serão entre os pares envolvidos. Neste caso o modelo é totalmente escalável. Se o administrador central for também o responsável pela busca de recursos nos pares armazenados no diretório, existirá uma sobrecarga maior

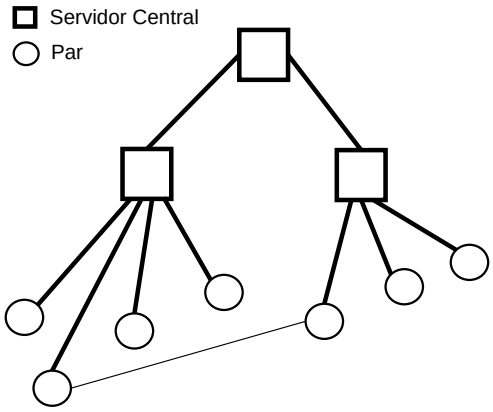


Figura 3.2: Modelo Centrado no Usuário.

porque todas as buscas serão feitas utilizando este administrador. Neste caso, o modelo fica pouco escalável e será necessário criar réplicas desse administrador, como no caso das aplicações Kazaa [kaz] ou Napster [nap].

Ingresso na rede. Para um par se conectar à rede, deve primeiro contatar o servidor central (com endereço conhecido) e pedir permissão para se conectar. Quando o administrador central permite o ingresso, este registra no seu diretório o novo par disponibilizando sua conexão com os outros pares da rede. Uma vez registrado, um par pode procurar no diretório por pares que correspondam aos critérios de busca dados pelo diretório. Devemos deixar claro que esse critério será a característica do par e não os recursos que ele disponibiliza. A informação recebida será os endereços desses pares e a transferência de informação será sempre entre os pares envolvidos e não afetará o administrador central.

Buscas na rede. A busca por pares conectados está implícita no diretório do administrador central, mas a busca por recursos terá que ser parecida com a do modelo atômico, ou seja, pela propagação da busca. Uma alternativa é ter diretórios dos recursos disponibilizados pelos pares da rede, como veremos na seção seguinte.

3.1.3 Modelo Centrado nos Dados

Esse modelo é muito parecido com o modelo centrado no usuário. A diferença principal é que o servidor mantém um índice dos recursos ao invés de usuários. Nesse modelo, as políticas de segurança e regras de conexão com os pares são mais estritas que no modelo centrado no usuário. Nem sempre é permitido o acesso aos recursos, especialmente no âmbito empresarial.

A escalabilidade desse modelo é menor que a do Centrado nos Usuários devido à quantidade de recursos disponíveis ser muito maior que a quantidade de pares. As mensagens de atualização de recursos disponíveis e as de busca por recursos podem sobrecarregar o administrador central provocando uma queda no desempenho. Como no caso do modelo centrado nos usuários, uma alternativa é usar réplicas destes administradores de modo a dispersar as mensagens de ingresso e busca.

Ingresso na rede Para o ingresso de um par nessa rede, é necessário verificar a segurança uma vez que alguns recursos podem não estar disponíveis para certos pares. De modo similar ao modelo centrado nos usuários, o par se contata com o administrador central pedindo permissão para se conectar. Quando o administrador central permite o ingresso, este registra no seu diretório todos os recursos disponibilizados pelo par que esta ingressando e as políticas de segurança desses recursos.

Buscas na rede Para buscas por recursos o par envia uma mensagem de busca para o administrador central. Este por sua vez procura os recursos disponíveis no seu diretório e devolve os endereços dos pares que o disponibilizam e que permitem o seu compartilhamento, segundo as políticas de segurança do recurso. O mapeamento entre recursos e endereços está no diretório do administrador central.

3.1.4 Modelo Estruturado

Este modelo foi um dos últimos a surgir no cenário dos modelos *Par-a-Par*. A idéia principal é a de construir redes estruturadas de modo a melhorar a eficiência das buscas por recursos nos pares da rede que o armazenam. Esse tipo de modelo é chamado de redes sobrepostas (*overlay networks*), veja Figura 3.3. Com isso, tenta-se evitar que a busca por um recurso seja propagada sem controle em pares que não têm o recurso, como no caso do modelo atômico.

Existem diferentes estruturas criadas para armazenar e procurar recursos em redes *Par-a-Par*, tais como: listas distribuídas [AS03], árvores distribuídas [Cos97] e tabelas de *hash* distribuídas [SMK+03].

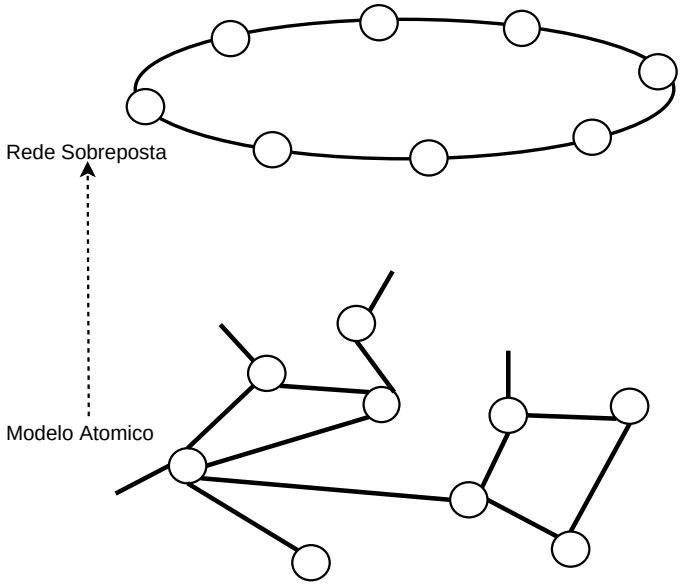


Figura 3.3: Modelo Estruturado.

A escalabilidade deste modelo, segundo as simulações feitas [RGRK04, SMK+03, RFH+01] foi demonstrada para milhares de pares unidos à rede. Mesmo assim ainda não existem experimentos para casos onde a quantidade de usuários é parecida com as aplicações reais, ou seja centena de milhares.

Ingresso na rede. Da mesma forma que no modelo atômico, aqui também não existe um servidor central ao qual se conectar à rede e portanto se tem os mesmos problemas e alternativas de conexão apresentadas naquele modelo.

Buscas na rede. Devido à estrutura formada pelos pares, a procura por recursos sempre devolverá um resultado positivo caso exista o recurso. A idéia é que o caminho seguido (chamado de roteamento) pela mensagem para procurar o par que armazena o recurso é o mais eficiente possível. 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).

3.2 Exemplos de Serviços

Mostramos na Tabela 3.1 algumas das aplicações *Par-a-Par* mais conhecidas.

Nome da Aplicação	Domínio	Arquitetura
ICQ	Serviço de mensagens	Centrado no usuário
Jabber	Serviço de mensagens	Centrado nos dados
Napster	Compartilhamento e transferência de arquivos MP3	Centrado nos dados/usuário
Gnutella	Compartilhamento e transferência de arquivos	Atômico
Freenet	Persistência de arquivos	Atômico
Groove	Ambiente de trabalho colaborativo	Centrado nos dados
JXTA	Persistência, mensagens, trabalho colaborativo, etc.	Atômico
Chord	Persistência, indexação de informações, trabalho colaborativo, etc.	Estruturado
CAN	Persistência, indexação de informações, trabalho colaborativo, etc.	Estruturado
BitTorrent	Compartilhamento e transferência de arquivos	Centrado nos dados
MSN	Serviço de mensagens	Centrado no usuário
Kazaa	Compartilhamento e transferência de arquivos	Centrado nos dados/usuário

Tabela 3.1: Aplicações *Par-a-Par*.

Capítulo 4

Localização Eficiente de Recursos

No âmbito das redes *Par-a-Par* as aplicações estão baseadas em características como: armazenamento, seleção do servidor mais próximo, buscas, autenticação, entre outras [SMK+03]. Nesses últimos anos, diversas pesquisas se concentraram no principal problema de todos os sistemas *Par-a-Par*, que é a localização eficiente do membro que armazena o recurso procurado.

Para resolver o problema acima, diversas estruturas distribuídas foram criadas (ver Seção 3.1.4), mas a tabela de *hash* distribuída teve efeitos revolucionários no desempenho e escalabilidade. A tabela de *hash* distribuída é uma estrutura que permite a realização das funções de uma tabela de *hash* normal, ou seja, nela pode-se armazenar um valor dada uma chave e procurar por esse valor utilizando a chave [Tan02].

Uma das características importantes sobre as tabelas de *hash* distribuídas é que o armazenamento e as operações de busca são distribuídas entre as máquinas que pertencem à rede *Par-a-Par*. Diferentemente das arquiteturas cliente/servidor existentes, os membros podem se unir e sair da rede livremente. Apesar do caos evidente que poderia acontecer com essas mudanças na estrutura da rede, o funcionamento da tabela de *hash* distribuída garante a qualidade de serviço nas operações de busca.

4.1 Características da Tabela de *Hash* Distribuída

As tabelas de *hash* distribuídas apresentam algumas características como:

- **Balanceamento de Carga.** É utilizada uma função de *hash* para distribuir igualmente as chaves a serem armazenadas entre os pares da tabela

de *hash* distribuída. Com isso, a probabilidade que um par fique sobrecarregado é muito baixa.

- **Descentralização.** Seguindo as propriedades das redes *Par-a-Par*, nenhum par nesta tabela é mais importante que outro, isso implica na melhoria da robustez comparado com alternativas baseadas em servidores devido à pouca organização deste tipo de redes.

- **Escalabilidade.** O custo das buscas só aumenta logaritmicamente com a quantidade de pares pertencentes à rede. Essa quantidade pode ser considerada constante mesmo para aplicações com um número muito grande de pares. Portanto, qualquer aplicação que seja baseada na tabela de *hash* distribuída pode ser desenvolvida sem se preocupar em colocar restrições que afetem o crescimento da rede.

- **Disponibilidade.** Essa característica impõe que toda chave será sempre encontrada, inclusive se o sistema está em um estado de mudança contínua. A tabela de *hash* distribuída preocupa-se com as entradas e saídas aleatórias dos pares da rede.

- **Flexibilidade de nomes.** As chaves a serem criadas pelos usuários da tabela de *hash* distribuída não precisam ter um formato específico.

- **Eficiência.** A tabela de *hash* distribuída provê mecanismos para o ingresso/saída de um par e para encontrar uma chave armazenada. Esse mecanismo somente precisa de uma troca de mensagens de ordem logarítmica em relação à quantidade de pares pertencentes à rede.

4.2 Estrutura de anel

A estrutura das tabelas de *hash* distribuídas pode ser vista como uma variação de uma lista circular duplamente encadeada onde cada nó dessa lista corresponde a um par da rede. Como a estrutura é circular, ela é chamada de anel. Na Figura 4.1 podemos observar uma estrutura de anel já formada com oito pares.

Cada par pertencente à estrutura têm uma informação única que o distingue, por exemplo a união do IP com o número da porta onde está escutando as requisições. A essa informação única é aplicada uma função de *hash* que devolve um identificador único (se a informação do par é única, o número gerado pela função terá uma alta probabilidade de ser único) o qual é inserido na

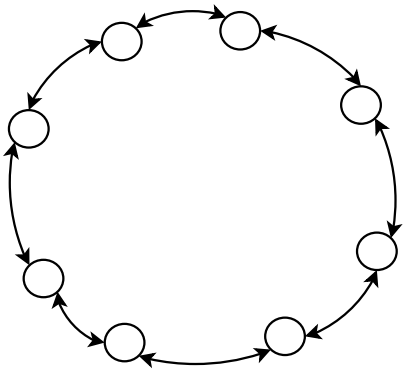
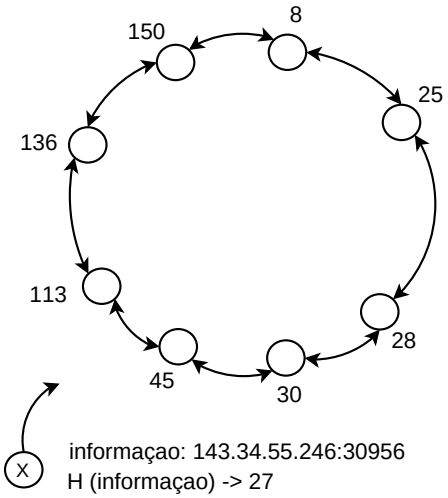
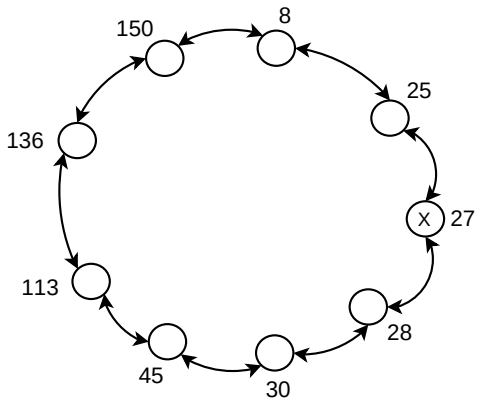


Figura 4.1: Estrutura de anel com oito pares.



(a) Transformação da informação do par X



(b) Inserção do Par X na estrutura

Figura 4.2: Exemplo de uma tabela de *hash* distribuída.

estrutura de forma que o anel fique sempre ordenado de forma crescente pelos números. Na Figura 4.2(a) podemos observar que ao par X que está ingressando é dado o identificador 27, o qual será inserido entre os identificadores 25 e 28 (Figura 4.2(b)).

Sobre a responsabilidade dos pares a respeito das chaves armazenadas, cada par da estrutura será responsável por um intervalo de chaves, compre-

endido desde o identificador do predecessor, sem contar com ele, até o seu identificador. Por exemplo, na Figura 4.2(a) o par com identificador 25 será responsável pelas chaves 9 (o primeiro número após o seu predecessor) até o 25.

A transformação de uma chave em um número, que servirá para conhecer que par será responsável por essa chave, é feita de maneira análoga ao ingresso dos pares. À chave a ser armazenada é aplicada uma função de *hash* a qual devolverá o número que será armazenado em algum par da estrutura responsável pela chave. Por exemplo, na estrutura como da Figura 4.2(a), uma chave com valor 32 será armazenada no par com identificador 45.

No caso de colisões das chaves, é a implementação da tabela de *hash* distribuída a responsável por administrar esse problema. Geralmente, as implementações inserem informações adicionais que permitem que a chave a inserir seja única com uma alta probabilidade. Logo, estruturas internas da tabela de *hash* distribuída possuem referências a essas colisões. Por exemplo, se tivermos uma chave X a ser inserida duas vezes, a implementação insere na tabela X, $X1$ e $X2$ ($X1$ e $X2$ são as chaves criadas a partir de X, que contém informações adicionais). Finalmente, o par responsável por X terá também referências às chaves $X1$ e $X2$.

4.3 Implementações

Como mencionamos no começo do capítulo, a estrutura em anel é duplamente encadeada, ou seja, cada par tem ponteiros para seu predecessor e para seu sucessor. Uma busca por uma chave nesse tipo de estrutura, portanto, requer tempo de ordem linear (a busca passará por cada par seguindo o sucessor até localizar o par responsável pela chave). Para que a eficiência de uma busca nas tabelas de *hash* distribuídas seja de ordem logarítmica, precisamos de uma estrutura adicional, chamada de tabela de roteamento, que permite conhecer pares que estão mais a frente que o sucessor imediato. Mostraremos a seguir duas implementações da tabela de *hash* distribuída: a primeira baseia sua tabela de roteamento em uma lista com ponteiros que seguem uma fórmula matemática e a segunda em uma estrutura de árvore de prefixos.

4.3.1 Chord

Chord [Cho] é uma implementação na linguagem C++ de uma tabela de *hash* distribuída feita nos laboratórios do MIT. O protocolo especifica como localizar

as chaves entre os pares da rede, como novos pares se unem à rede e, finalmente, como os pares saem da rede.

Tabela de Roteamento Suponhamos que temos, em um dado momento, a seguinte estrutura de anel mostrada na Figura 4.3. Cada par com identificador p tem um tabela de roteamento local que aponta para pares que estão mais na frente que ele no anel e que ajuda na localização eficiente dos recursos. Esta tabela de roteamento, que é chamada de *fingers* pelo Chord, é definida da seguinte maneira (Ver Tabela 4.1):

- Há no máximo $\log n$ ponteiros registrados (n é a quantidade de pares na estrutura) evitando com isso um excesso de informação. Se tivéssemos todos os pares registrados tornaríamos o protocolo pouco escalável.
- Cada entrada i da tabela de roteamento de p contém o identificador do primeiro par no anel cujo valor seja maior ou igual a $p+2^i$ e menor que o identificador do último par da tabela de *hash* distribuída. Por exemplo, na Figura 4.3, a entrada $i = 3$ tem como identificador o par 45 que é o primeiro par encontrado cujo valor é maior que 33 ($p+2^i = 25+8 = 33$).

Entrada	Definição
0	ponteiro ao successor de p
i	ponteiro ao primeiro par s no anel tal que $p + 2^i \leq s$, $1 \leq i \leq log_2$(id do último par - p)

Tabela 4.1: Definição da tabela de roteamento de um par p.

Busca por um recurso. A busca por um recurso dada uma chave é feita da seguinte maneira. Quando um par envia uma mensagem em busca de uma chave, é aplicada a ela uma função de *hash* que devolve um valor v, $h(chave) = v$. A seguir, a tabela de roteamento fornece o identificador $p < v$ (onde p é o valor mais próximo de v conhecido pela tabela). A partir de p, o processo se repete de maneira recursiva, passando essa mensagem até se encontrar o valor v. Este processo é completado com $O(\log n)$ troca de mensagens.

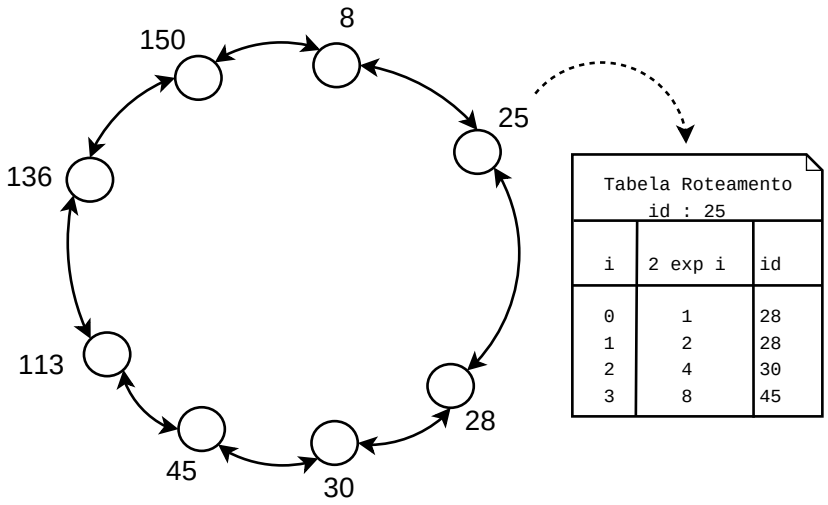


Figura 4.3: Exemplo da tabela de roteamento do par número 25.

Para dar um exemplo consideremos a seguinte situação: suponha que um par identificado com o número 25 precisa procurar a chave cuja função de *hash* devolveu o valor 122. Os passos seguintes se manifestam na Figura 4.4.

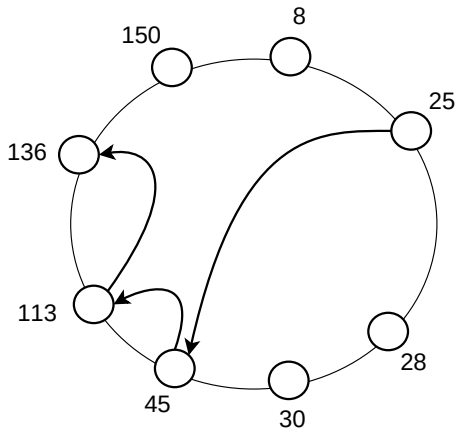


Figura 4.4: O par número 25 procura pela chave 122.

O par 25 verá, em sua tabela de roteamento, que existe um par com identificador mais próximo à chave procurada, neste caso o par 45. A busca será repassada ao par 45 que por sua vez verá em sua tabela de roteamento

o par mais próximo à chave procurada, que é o par 113. A busca é repassada novamente com o qual se chegará ao par 136, responsável por manter as chaves do 114 ao 136.

A quantidade de troca de mensagens utilizada para executar essa busca é $O(\log n)$, pois as entradas das tabelas de roteamento evitam que uma mensagem passe por todos os pares da estrutura [CJK+01].

Ingresso de um par na tabela de *hash* Em um ambiente dinâmico como as redes *Par-a-Par*, o ingresso de um par na estrutura pode acontecer a qualquer momento. Para manter as propriedades da tabela de *hash* distribuída, faz-se necessário levar em conta as seguintes considerações:

- Alocação do novo par.

  Primeiro deve-se encontrar o lugar exato onde incluir o novo par. Para isso, faz-se uma busca (perguntando a qualquer par da tabela de *hash* distribuída) pelo identificador do novo par. O ponto final do caminho traçado por essa busca permitirá saber qual é o par mais próximo da posição onde teremos que inserir o novo par (Na Figura 4.2(a), o par com identificador 28 é o mais próximo do novo par). A quantidade de troca de mensagens necessária para executar a alocação é $O(\log n)$, que seria a busca da posição mais a inserção do par, que leva tempo constante.

- Atualizar os registros dos outros pares já existentes na tabela de *hash* distribuída.

  A primeira atualização é feita sobre o par sucessor s e o par predecessor p do novo par ingressado, que chamaremos de n. n contata o seu sucessor s indicando que atualize seu ponteiro predecessor (que agora apontará para n). O mesmo processo é feito para o predecessor de p, o ponteiro sucessor de p agora apontará para n.

  A segunda atualização corresponde às entradas das tabelas de roteamento dos outros pares para que estes tenham conhecimento do novo par que ingressou. Para cada entrada é feita uma procura pelo valor, segundo a fórmula da Tabela 4.1, sem considerar o valor que a tabela de roteamento contém. Na Figura 4.5, podemos observar que a entrada $i = 1$ depois de executar este processo vai apontar para o par 27 e não para o que tem atualmente (identificador 28). Lembremos que segundo as regras da tabela: $n + 2^i = 25 + 2^1 = 27$

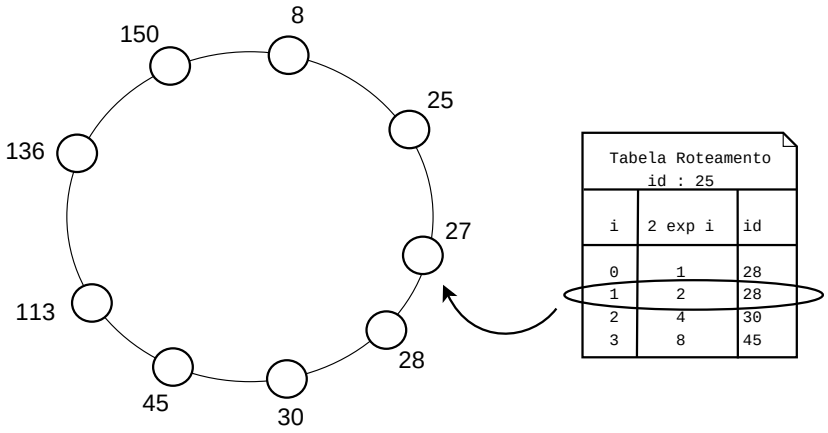


Figura 4.5: Atualização da tabela de roteamento identificando o par 27.

As duas atualizações apresentadas são executadas com uma troca de mensagens de ordem logarítmica [SMK+03].

- Transferência das chaves ao novo par

  Esta última operação move as chaves (identificadas como k) que agora serão de responsabilidade do novo par ingressado. Somente se precisa extrair do sucessor imediato do novo par as chaves do intervalo entre o identificador do novo par até a do identificador do sucessor. Este processo é mostrado na Figura 4.6 na qual podemos observar que as chaves 27 e 26 são transferidas para o par 27.

Saída de um par da tabela de *hash* Da mesma forma que para o ingresso de um par na estrutura, para a saída de um par, que pode ser planejada ou não, devemos considerar os seguintes aspectos:

- Atualizar os registros dos outros pares já existentes na tabela de *hash* distribuída.

  É necessário atualizar a tabela de roteamento dos outros pares para que estes tenham conhecimento do par que saiu da estrutura. Para isso é utilizado o mesmo processo de atualização quando um par ingressa na rede. Primeiro se irá procurar o identificador do par para cada entrada da tabela de roteamento. Essa atualização é executada com uma troca de mensagens de ordem logarítmica [SMK+03].

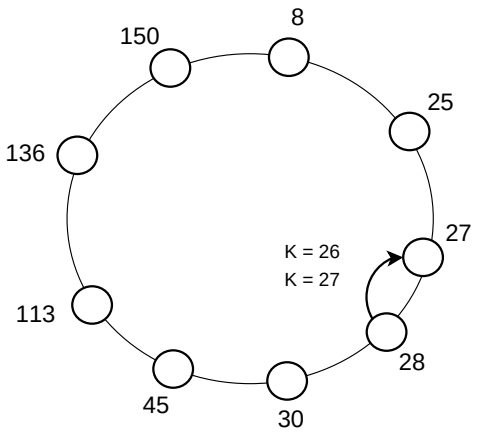


Figura 4.6: Transferência das chaves do par 28 para o par 27.

- Transferência das chaves ao sucessor do par que saiu.

  Esta última operação move as chaves do par que saiu da estrutura para o sucessor imediato dele, já que agora serão de responsabilidade do sucessor.

4.3.2 Bamboo

Bamboo é uma implementação da tabela de *hash* distribuída na linguagem Java desenvolvida na Universidade da California em Berkeley. A equipe do Bamboo direcionou os esforços de desenvolvimento no melhoramento do desempenho e estabilidade da rede quando os pares entram e saem dela com uma alta probabilidade [RGRK04]. O protocolo descreve como achar as chaves entre os pares da rede, como novos pares se unem à rede e o processo de atualização da estrutura.

Tabela de Roteamento A tabela de roteamento do Bamboo que, como no caso do Chord, tem ponteiros a pares que estão na frente no anel e ajudam a melhorar a eficiência nas buscas por uma chave, utiliza uma versão da tabela muito parecida como utilizado pelo algoritmo Pastry [RD01] que é baseado em uma estrutura de árvore de prefixos.

Essa tabela contém os identificadores de outros pares pertencentes à tabela de *hash* distribuída e consiste em uma matriz com $\log n$ filas e 2^b colunas (começando de zero) no qual 2^b é a base numérica do identificador, geralmente

com $b = 2$. Com isso, qualquer número que esteja na base 10 (ou outra base) será transformado na base 4.

Na Figura 4.7(a) temos a tabela de roteamento do par com identificador 10123102 (base 4) ou 19410 (base 10). Como podemos observar, cada célula $fila, coluna$ da tabela de roteamento corresponde a um identificador de um par. Este identificador está dividido em três partes:

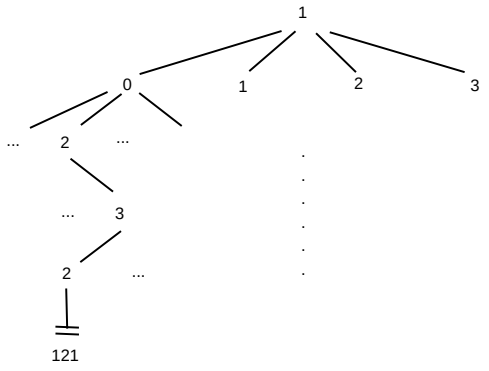
1. um prefixo do identificador formado pela união de todas as células sombreadas até a $fila - 1$
2. o valor da *coluna*
3. um sufixo que, na união com os outros dois valores acima, corresponde a um identificador válido de um par pertencente à estrutura.

Se essa célula estiver vazia quer dizer que a tabela não conhece nenhum par com esse prefixo. Para dar um exemplo, vemos que na Figura 4.7(a) e 4.7(b), a célula da fila 4 coluna 2 têm como prefixo 1023, como sufixo 121 e a união do prefixo-coluna-sufixo representa a o par com identificador 10232121. No caso da ultima fila, podemos observar que está vazia, significando que não existem pares com o prefixo 1023310.

Tabela de Roteamento
Identificador: 1 0 2 3 3 1 0 2

	0	1	2	3
0		1		
1	0	1-1-301233	1-2-230203	1-3-021022
2	10-0-31203	10-1-32102	2	10-3-23302
3	102-0-0230	102-1-1302	102-2-2302	3
4	1023-0-322	1023-1-000	1023-2-121	3
5	10233-0-01	1	10233-2-32	
6	0		102331-2-0	
7			2	
8				

(a) Tabela de roteamento para o par com identificador 10233102



(b) Árvore de prefixos para o identificador 10232121

Figura 4.7: Exemplo de uma tabela de roteamento do pastry.

Busca por um recurso. A busca por um recurso é feita da seguinte forma. Quando um par envia uma mensagem em busca de uma chave, será aplicada uma função de *hash* à chave que devolverá um valor v. A tabela de roteamento devolverá então o identificador do par que tenha o prefixo mais longo comparado com v (vamos supor que esse par tem como valor p). A partir de p, o processo se repete de maneira recursiva, passando essa mensagem até se encontrar o valor v. Para efetuar esta busca é necessário $O(\log n)$ troca de mensagens.

Por exemplo, vamos supor que estamos procurando pelo valor $v = 10232123$. A tabela de roteamento da Figura 4.7(a) devolverá $p = 10232121$ (segundo a Figura 4.7(b) o prefixo conhecido mais longo é 10232). A partir de p esse processo se repete até localizar o valor 10232123.

Ingresso de um par na tabela de *hash* Como acontece com Chord, faz-se necessário levar em conta algumas considerações para o ingresso de um par na estrutura do Bamboo:

- Alocação do novo par.

  O lugar na tabela onde será inserido segue as mesmas regras que no Chord, ou seja, faz-se uma busca pelo identificador do novo par até localizar o valor do sucessor mais próximo a esse identificador. Essa será a posição correta na tabela de *hash* distribuída onde será inserido o novo par. A diferença com Chord está em que, neste processo, ao novo par é dado uma tabela de roteamento inicial, com valores muito parecidos com a tabela de roteamento do seu sucessor.

- Atualizar os registros dos outros pares já existentes na tabela de *Hash* distribuída.

  A atualização é dada da seguinte forma. Primeiro se escolhe de forma aleatória uma célula $(fila, coluna)$ da tabela de roteamento de um par. Essa célula têm associado um valor v (dado pelo prefixo da célula mais o valor da coluna) o qual será procurado na tabela de *hash* distribuída da seguinte maneira: a mensagem de busca, por esse valor v, é repassada ao primeiro identificador do par desta tabela de roteamento que pertença à mesma $fila$ da célula e $coluna = i$ com $(coluna > i \geq 0)$.

  Para dar um exemplo, vamos supor que de forma aleatória escolhemos atualizar a célula (5,3) da tabela de roteamento da Figura 4.7(a). Segundo anteriormente mencionado, o valor v procurado será um par que

tenha prefixo 102333 (prefixo 10233 mais a *coluna* = 3). Agora, para encontrar um par ao qual repassar a mensagem de busca teremos que ir diminuindo o valor da coluna. A célula (5,2) contém o par 10233232. No caso dessa célula estar vazia a mensagem será repassada ao par 10233001, ou seja, para a célula (5,0).

Saída de um par da tabela de *hash* Quando um par sai da estrutura faz-se necessário atualizar as tabelas de roteamento dos outros pares já existentes na tabela de *hash* distribuída. Para isso, utiliza-se o mesmo processo de atualização de registros, ou seja, escolhe-se uma célula que tem associado um valor e procura-se pelo valor como definido anteriormente na atualização quando ingressa um novo par.

Capítulo 5

Protocolo de Interligação

Esta seção apresenta o protocolo que permitirá montar uma estrutura de pares que seja eficiente na comunicação. Chamaremos este protocolo de *Protocolo de Interligação*. Esse protocolo, baseado no modelo atômico (*vide* Seção 3.1.1), visa interligar os pares mais próximos entre si em termos de latência. Antes de descrever os detalhes do protocolo, vejamos quais são os principais motivos pelo qual se faz necessário criá-lo.

Primeiro, existe uma necessidade no InteGrade de interligar os GRMs dos diferentes aglomerados. Com isso, no caso em que uma tarefa não possa ser realizada localmente, teremos a possibilidade de requisitar a execução desta tarefa a outros GRMs.

Segundo, nos protocolos *Par-a-Par* existentes, como no caso de *Gnutella* [gnub] ou do *Gisp* [Kat02] do *JXTA* [jxt], a informação que um par disponibiliza (como por exemplo um arquivo de música) somente é encontrada em alguns pares. Em nosso caso, as informações que um par compartilha (como sua quantidade de memória RAM disponível) devem estar na maioria dos pares.

Terceiro, na maioria das redes *Par-a-Par* que utilizam o modelo atômico, a estrutura e formação dos pares é criada de forma aleatória em relação a seus vizinhos[1]. Para o nosso trabalho, o tempo de latência entre um par e seus vizinhos é um requisito importante, que não é considerado nos protocolos desenvolvidos até agora.

Veremos agora como o nosso protocolo resolve os problemas anteriormente mencionados. É importante destacar que esse protocolo, desenvolvido para redes *Par-a-Par*, é uma generalização da necessidade do InteGrade de interligar os GRMs. Com isso, se queremos incluir o protocolo no InteGrade

[1] A vizinhança de um par p é definida como os pares que p conhece.

teremos que tratar cada GRM como se fosse um par.

5.1 Visão Geral

Como mencionado no modelo atômico da Seção 3.1.1, as redes *Par-a-Par* não têm uma estrutura de conexão estabelecida, ou seja, não seguem nenhum padrão definido, como mostrado na Figura 5.1. Para que um par possa melhorar o desempenho no envio e recepção das mensagens, se faz necessário que ele se conecte com o par mais próximo considerando o tempo de latência e não com outro qualquer. No caso do InteGrade, a melhora no desempenho da comunicação serviria para que um GRM possa repassar, de forma eficiente, uma tarefa que não pôde ser executada localmente. A seguir veremos como resolver esse problema usando a função de roteamento da camada de rede.

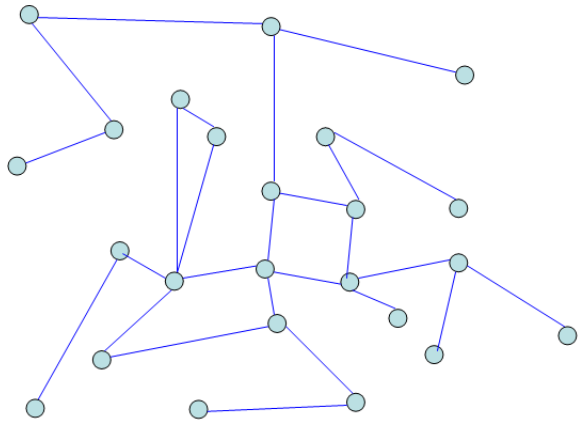


Figura 5.1: Estrutura de uma rede *Par-a-Par* geral.

5.1.1 Usando e Armazenando a Informação dos Roteadores

A camada de rede tem como função entregar pacotes de um lugar a outro através de uma infra-estrutura de redes interconectadas [Sta03]. Para isso, os protocolos usados nesta camada fazem a determinação de caminhos (ou

roteamento) entre destinos o que permite estabelecer a rota de preferência para o envio de pacotes.

A compreensão da rota seguida por um pacote pode ajudar a descobrir que pares estão próximos em termos de latência. Conforme ilustrado na Figura 5.2, se o par $p1$ envia uma mensagem para o par $p2$, 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 par $p3$, que não é conhecido e que está mais perto de $p1$, em termos de latência, do que $p2$. Se pudéssemos conhecer $p3$, $p1$ poderia se comunicar com $p3$, assim, a comunicação entre eles provavelmente seria mais rápida do que a comunicação entre $p1$ e $p2$.

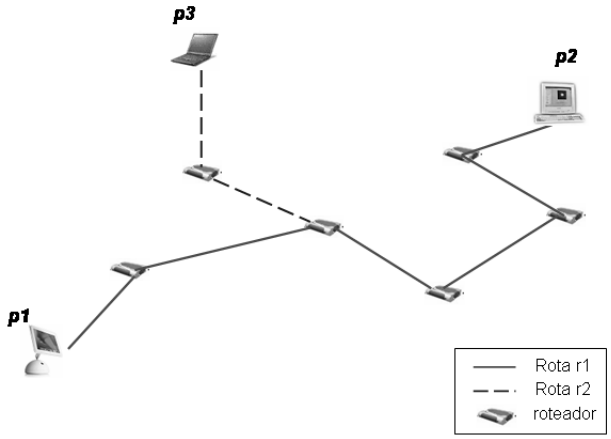


Figura 5.2: Rota da mensagem entre dois pares.

Nosso protocolo usa e armazena as informações de latência e das rotas da seguinte maneira:

- Informações sobre cada roteador presente em uma rota entre dois pares serão armazenadas na Tabela de *Hash* Distribuída (DHT). A DHT armazenará em uma estrutura, que chamaremos de "objeto roteador", a latência e o identificador do par mais próximo a ele.
- Usaremos as informações armazenadas nos objetos roteadores no ingresso de um par na rede, detalhado na Seção 5.2.1.

5.1.2 Repositório Local de Pares

No Protocolo de Interligação, o repositório é um componente muito importante, já que com ele podemos armazenar conhecimento sobre os vizinhos de um par. Esse conhecimento corresponde a uma lista de identificadores dos pares, ordenada em ordem crescente de latência, armazenada num arquivo local ao par. Os pares, junto com sua latência, são atualizados periodicamente pelo processo de atualização do repositório (vide Seção 5.2.3). O identificador do par deve ser único e depende do implementador escolher de que tipo vai ser. Geralmente usa-se o endereço IP e a porta na qual o par recebe as requisições.

PeerID	Latência(ms)
217.73.145.120:2411	10.01
164.109.41.38:1475	10.33
69.2.200.183:3352	12.87
212.40.5.72:2463	12.87
209.203.253.95:2674	12.98
198.65.117.133:4213	13.53
83.97.42.2:80	15.08
205.217.153.53:1113	15.48
195.95.30.170:3442	17.65

Tabela 5.1: Exemplo de um Repositório local de Pares

Na Tabela 5.1, vemos que o repositório de um par proporciona informação sobre pares vizinhos (endereço IP : porta) e a latência entre eles. No caso do par se desconectar da rede e voltar a entrar, o repositório será muito útil na obtenção dos seus antigos vizinhos. Na implementação do repositório, deveria se usar uma quantidade de não mais que 10 vizinhos, que evita que a atualização periódica gere uma sobrecarga na rede pela quantidade de informação a atualizar.

5.2 Descrição do Protocolo

Para garantir a interligação usando a proximidade de latência, nosso protocolo deve considerar três situações do par $p1$: a entrada na rede, a saída da rede e as atualizações das referências dos outros pares com $p1$. A seguir detalharemos cada uma dessas situações.

5.2.1 Adicionando um Par à Rede

Segundo o detalhado na Seção 3.1.1, em um modelo arquitetônico atômico como o nosso protocolo, o par que quer ingressar na rede precisa conhecer "outros pares" quaisquer aonde se conectar. Para isso, podemos identificar duas opções:

- *É a primeira vez* que o par se conecta à rede: obteremos esses outros pares de uma fonte na Internet, como por exemplo uma página Web.
- *O par já tinha ingressado antes* e deseja voltar a se conectar na rede: obteremos esses outros pares do repositório local (*vide* Seção 5.1.2) que os armazena.

Primeira vez. A solução proposta é ter o endereço de alguns pares, mais estáveis, em um local fixo, por exemplo, uma página Web que todos conheçam. Estes pares "estáveis" devem ter como característica o fato de a probabilidade de saírem da rede ser baixa. Outra possibilidade é a busca através de *Broadcast* esperando que alguém atenda à requisição, mas isto é claramente não escalável.

Uma das características da solução usando a página Web é que ela deve conter uma quantidade pequena de pares (no máximo algumas dezenas), evitando com isso uma demora no processamento da página. Outra característica é que esta página não seja necessariamente única, ou seja, podem existir outras páginas organizadas por grupos, países, etc, que permitiria uma maior escalabilidade. Na Tabela 5.2 vemos um exemplo de uma possível página Web que armazena os endereços e portas onde os pares recebem as mensagens.

Já tinha ingressado. Neste caso, não devemos nos preocupar em procurar os pares na página Web descrita anteriormente. Agora simplesmente os obtemos do repositório de pares.

Na Figura 5.3, apresentamos o algoritmo de ingresso de um par $p1$ na rede. É importante destacar que este algoritmo é executado no par que está ingressando na rede. Detalharemos o seu funcionamento a seguir.

1. Obtemos uma quantidade constante de possíveis identificadores de pares com os quais o novo candidato ($p1$) poderia se conectar. Esses identificadores serão obtidos do processamento da página Web, se for a primeira vez que o novo candidato se conecta, ou do repositório local de pares, se já tinha ingressado antes (Linha 3 e 5).

PeerID	Porta
193.154.180.100	80
63.211.182.17	156
66.250.128.130	80
4.22.66.35	1225
69.31.132.42	678
209.203.253.95	2674
164.109.41.38	1475
195.95.30.170	3442
212.40.5.72	2463
217.73.145.120	2411
198.65.117.133	4213
83.97.42.2	80
69.2.200.183	3352
205.217.153.53	1113
209.59.152.158	80

Tabela 5.2: Lista de pares estáveis a ser armazenada em uma página Web.

```
1: INGRESSO (P1)
2: se (busca == "web")
3:     possíveis ← obter_pares_da_web();
4: caso contrário
5:     possíveis ← obter_pares_do_repositório();
6: escolhido ← obter_par_menor_latência(possíveis);
7: ip_roteadores ← roteadores_entre_p1_e_escolhido();
8: objetos_roteador ← dht.roteadores_entre_p1_e_escolhido(ip_roteadores);
9: para cada roteador contido em objetos_roteador {
10:     p2 ← roteador.par_mais_próximo();
11:    se (latência(p1, p2) < latência(p1, escolhido))
12:        escolhido ← p2;
13:    caso contrário
14:        dht.atualiza_roteador(roteador, escolhido);
15: }
16: devolve escolhido;
```

Figura 5.3: Algoritmo para o ingresso de um par na rede.

2. É escolhido o par com menor latência entre $p1$ e os pares obtidos acima, deixando registrado no repositório de pares de $p1$ as latências obtidas

(Linha 6). O registro dos pares no repositório segue certas regras que foram descritas na Seção 5.1.2.

É importante destacar que existem diversas formas de obter a latência. Em nosso caso, o protocolo utiliza o *ICMP (Internet Control Message Protocol)* [Pos81] através de seu comando *Ping*, que obtém, do teste de conexão, o tempo necessário para que uma mensagem atinja o seu destino e retorne à origem.

3. Com o mais próximo dos pares encontrados (o *escolhido*) usamos os recursos físicos de rede que permitirão encontrar o melhor caminho entre $p1$ (o par que esta se unindo à rede) e o *escolhido*. Para isto, temos que obter os endereços IP de todos os roteadores do caminho entre o $p1$ e o escolhido (Linha 7). Para a obtenção desses roteadores, podemos usar por exemplo, o comando *traceroute* do Unix, com o qual obteremos também a latência de $p1$ a cada roteador encontrado por este.

4. Para todos os endereços IP dos roteadores encontrados acima, consultamos na Tabela de *Hash* Distribuída se existe alguma informação sobre esses roteadores (Linha 8). O tipo de informação armazenado nos *objetos_roteador* foi explicado na Seção 5.1.1.

5. Obtido o par $p2$ armazenado no objeto *roteador* (Linha 10), verifica-se se a latência entre $p1$ e $p2$ – obtida da fórmula: latência ($p1$, *roteador*) + latência (*roteador*, $p2$) – é menor que a latência entre $p1$ e o *escolhido* (Linha 11). Caso seja menor, $p2$ será o novo *escolhido* por estar mais perto de $p1$ (Linha 12). Caso contrário, o *escolhido* está mais perto de $p1$ e, portanto, devemos atualizar o par armazenado no *roteador* (Linha 14).

6. Finalmente, o algoritmo devolve o *escolhido* com o qual $p1$ se conectará (Linha 16).

5.2.2 Saída de um Par da Rede

Neste processo, o par se desconecta da rede. A desconexão pode ser provocada por diversos motivos entre eles:

- Saída voluntária
- Saída por problemas técnicos

De acordo com Anderson et al. [ASSW03], devemos estar cientes de que a queda de um par é possível e provável. Então, quando um par sai da rede, o protocolo deve: (1) atualizar as referências que os pares conectados a ele tinham quando saiu e (2) atualizar os objetos *roteadores* que tinham armazenado informações sobre o par sendo desconectado. É importante destacar que essas atualizações são feitas periodicamente e não necessariamente no momento em que um par sai da rede. Por ser um processo que ocorre de forma independente, as atualizações serão mostradas na próxima seção.

5.2.3 Processo de Atualização

Nosso protocolo realiza o processo de atualização de referências periodicamente e independente de um par entrar ou sair da rede. Existem três tipos de atualizações que serão mostradas a seguir. As duas primeiras são as atualizações de referências que dizem respeito à estrutura lógica das conexões entre os pares e a última corresponde à atualização do repositório que diz respeito à forma de obter novos pares.

Atualização de Referências. A atualização de referências pode ser feita quando um par (o *escolhido*) entra ou sai da rede. De qualquer forma, caso um par p tenha como referência o *escolhido*, p é responsável por verificar, periodicamente, se o canal de comunicação entre eles está funcionando. Caso não exista uma comunicação, p terá que se conectar com o primeiro par do seu repositório de pares. Se não existir nenhuma referência no repositório, então ele terá que se conectar novamente usando o algoritmo para conectar-se à rede, explicado na Seção 5.2.1.

Atualização de Objetos Roteadores. A atualização dos objetos roteadores ocorre quando um par sai da rede. Aqui é necessário remover a referência contida no objeto roteador. Para isso, cada objeto roteador verificará se o par que mantém armazenado está funcionando (com o mesmo teste de conexão mostrado no item 2 da explicação do algoritmo de ingresso). Caso não exista comunicação, o par e o objeto roteador serão removidos da Tabela de *Hash* Distribuída.

Atualização do Repositório de Pares Essa atualização consiste em atualizar as referências aos pares do repositório que não estejam disponíveis e em obter novos pares. No primeiro caso, o protocolo terá que verificar, com um teste

de conexão, se os pares do repositório estão funcionando ou não. Caso o par esteja funcionando, é atualizada sua latência e caso não esteja funcionando, é eliminado do repositório. Para a verificação de conexão, tem-se que enviar e receber mensagens de todos os pares do repositório, o que pode resultar em uma saturação da largura de banda para aplicações com centenas de milhares de pares.

No segundo caso, ou seja, na obtenção de novos pares, pergunta-se a algum dos vizinhos tomado de forma aleatória se ele conhece outros pares (que não existam no repositório) que cumpram com o requisito de estar entre as primeiras T posições do repositório em termos de latência (T é um valor determinado pelo usuário do protocolo). Como vimos na Seção 5.1.2 o repositório é uma lista ordenada de pares pela latência. No caso de não encontrar nenhum par que esteja entre as primeiras T posições, propagaremos o pedido de novos pares aos vizinhos. Essa propagação terá uma profundidade determinada pelo valor de uma variável TTL (*Time to Live*) que diminuíra a cada propagação. Na figura 5.4, podemos observar o caso no qual TTL = 3. Os números dentro dos quadrados representam o valor do TTL que vai diminuindo em cada nível de propagação.

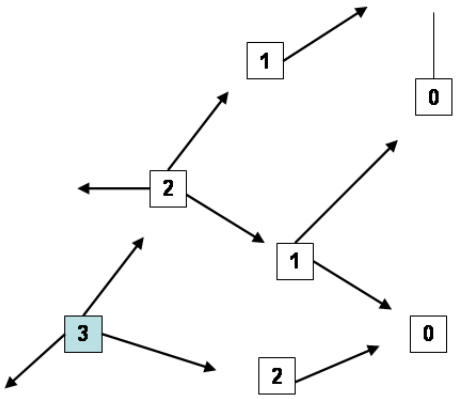


Figura 5.4: Propagação de uma mensagem pelos pares com o TTL = 3.

Um detalhe importante é como evitar a geração de ciclos entre os pares, ou seja, como evitar que a pergunta seja refeita para um mesmo par. Uma

alternativa é enviar na mensagem uma lista dos pares que já foram visitados. A outra é deixar que ciclos aconteçam e o par que fez a requisição descarte os que são iguais.

Capítulo 6

Protocolo de Localização

Esta seção apresenta o protocolo que servirá para localizar as informações disponibilizadas pelos pares. No caso específico do InteGrade, existe a necessidade de localizar nós de diferentes aglomerados que atendam aos requisitos para a execução de uma tarefa. Por exemplo, como mencionado na Seção 1.1, encontrar 64 máquinas que tenham um mínimo de 256 MB de RAM disponível para executar uma aplicação de multiplicação de matrizes.

Para resolver esse tipo de buscas, foi criado o Protocolo de Localização, que permite a busca de informações de forma a obter a maior quantidade de pares que satisfaçam um certo requisito baseado nas informações publicadas.

6.1 Tipos de Recursos

Antes de detalhar o protocolo, vejamos quais são os tipos de recursos publicados por um par que poderão ser localizados pelo protocolo:

- **Recursos estáticos**

  São recursos onde os atributos tendem a não mudar com o tempo, por exemplo, um documento PDF, um programa de multiplicação de matrizes, um arquivo de configuração do protocolo, etc.

- **Recursos dinâmicos**

  São recursos onde algum dos seus atributos mudam de valor freqüentemente, por exemplo, a memória RAM disponível, percentual livre do processador ou o espaço livre em disco rígido de um computador.

No primeiro caso, a busca pelos recursos é resolvida usando diretamente a Tabela de *Hash* Distribuída, ou seja, pode-se deixar, em pares distribuídos,

referências para os recursos a compartilhar. Para isso, pode se criar uma chave com os dados relevantes do recurso, como o nome do arquivo, e armazená-la na tabela. O segundo caso é mais complicado. Como a informação pode mudar sem um padrão definido, temos que ter alguma forma de poder acessá-la. Mostraremos a seguir quatro diferentes propostas de solução.

6.2 Alternativas de Localização

Existem diferentes alternativas para localizar os recursos disponibilizados pelos integrantes de uma rede *Par-a-Par*.

1. Uma primeira alternativa seria deixar um par como administrador central que controlasse a informação dinâmica de todos os pares, ou seja, que recebesse uma requisição e devolvesse uma resposta. Isso é impraticável em uma rede muito grande pois não é escalável, devido à sobrecarga do par que mantém a informação.

2. A segunda alternativa seria perguntar à vizinhança do par se eles conhecem o recurso. Como mostrado na Seção 5, com a estrutura formada pelos pares, asseguramos que a troca de mensagens será feita entre pares com uma latência mínima, o que permite um bom desempenho. Neste caso, seria preciso levar em conta as seguintes considerações:

   - A vizinhança do par pode ser obtida através do repositório de pares, ou perguntando diretamente ao par conhecido mais próximo. A partir deste vizinho a mensagem de busca pode ser propagada.
   - No caso da propagação da mensagem, temos que evitar os ciclos gerados quando se pergunta novamente a um par e a propagação sem controle consumindo a largura de banda de forma desnecessária.

   O problema de perguntar à vizinhança é que não podemos ter certeza se o recurso existe. A propagação tem que ter limites para evitar congestionamento da rede pelas mensagens trocadas e para o bom desempenho (não podemos esperar que a busca seja repassada a todos os pares). Com isso, se o recurso existe além do nível da propagação, pode não ser encontrado.

3. A terceira alternativa, estudada por diferentes grupos de pesquisa [Lui04] [Jun04], seria a de se manter uma árvore distribuída onde cada nó da árvore representaria um intervalo de valores (como uma árvore B [Knu73])

de disponibilidade de recursos, o que permitiria uma busca eficiente dos recursos. Cada intervalo seria uma chave armazenada na tabela de *Hash* distribuída. A deficiência dessa estrutura ocorre quando todos os valores pertencem somente a um intervalo, nesse caso teremos uma sobrecarga no par responsável por ele, gerando uma arquitetura cliente/servidor. Desse modo o desempenho cai drasticamente, tornando o protocolo pouco escalável.

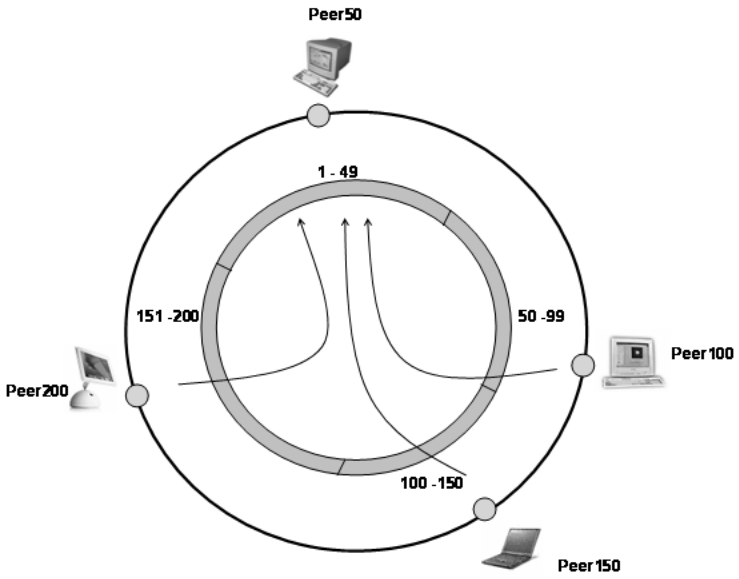


Figura 6.1: Todos os pares atualizam num só par gerando uma arquitetura Cliente-Servidor.

Essa solução está ilustrada na Figura 6.1, onde podemos observar que o par com identificador 50 é o responsável pelo intervalo de valores de memória RAM disponível entre 1 e 49 MB. No caso de todos os pares estarem dentro desse intervalo, teríamos problemas de desempenho na resposta à requisições devido à sobrecarga em um único nó.

Além disso, existe um problema relacionado à vizinhança dos pares. A resposta que teríamos de uma busca nesse intervalo seriam pares que não necessariamente estariam perto do cliente que fez a pergunta. Com isso,

a troca de mensagens poderia ter um custo maior.

4. A quarta alternativa propõe criar uma estrutura baseada em uma lista distribuída ordenada para o armazenamento das informações dinâmicas e que poderá ser usada na busca eficiente de recursos [RR05]. Esta alternativa, que utiliza também a tabela de *Hash* distribuída, pode diminuir a quantidade de objetos armazenados, em comparação com a solução da árvore, e portanto diminuir também a quantidade de mensagens trocadas para manter esse tipo de estrutura. Esta estrutura será detalhada na próxima seção.

6.3 Lista Distribuída

Como mencionado anteriormente, a última alternativa utiliza a tabela de *hash* distribuída para o armazenamento dos recursos a serem localizados. Entretanto, um dos problemas das tabelas de *hash* distribuídas 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 superior e um inferior. Para resolver este problema, propomos um novo mecanismo que é simples, eficiente e escalável. Esse mecanismo é baseado em uma lista distribuída ordenada pelos valores dos identificadores, que denominamos Lista para Busca por Intervalos (LBI).

6.3.1 O Problema das Buscas nas tabelas de *hash* distribuídas

Nas tabelas de *hash* distribuídas existem dois tipos de buscas possíveis: busca simples e busca por intervalo.

Uma busca simples é definida como a obtenção de um dado específico armazenado pela tabela de *hash* distribuída, a partir do identificador deste dado. Esta funcionalidade é encontrada explicitamente nas implementações das tabelas de *hash* distribuídas. Existem algumas propostas, como o uso de elementos com formato (atributo,valor) [BBK02] ou a decomposição do identificador [HHB+03], que manipulam os identificadores de forma a melhorar a eficiência das buscas. Um exemplo onde este tipo de busca é aplicado são as Grades computacionais como o InteGrade [Int, GKG+04], onde um usuário pode estar interessado em localizar computadores com certos recursos disponíveis, como mostrado na Figura 6.2.

Obter máquinas onde:

osName = Linux &&
processorMhz = 500;

Figura 6.2: Exemplo de uma busca simples com uma combinação de elementos atributo-valor.

Uma busca por intervalo consiste em encontrar o conjunto de todos os dados que estão contidos em um determinado intervalo de identificadores. Por exemplo, suponha que um usuário deseja obter uma lista de computadores que possuam uma quantidade mínima de memória RAM. Este tipo de busca não é oferecida diretamente pelas tabelas de *hash* distribuídas e é representada como uma desigualdade no elemento (atributo-valor), mostrado em negrito na Figura 6.3.

Obter máquinas onde:

osName = Linux &&
availableRAM $\geq$ 512;

Figura 6.3: Exemplo de buscas por intervalo de valores.

Para resolver em parte esse problema, Gao [GS04] apresentou duas propostas. A primeira consiste em registrar na tabela de *hash* distribuída um identificador para cada valor contido no intervalo. Por exemplo, se o identificador é "RAM" e o intervalo está entre r_{inicio} e r_{fim}, a quantidade de identificadores será $d = fim - inicio$ representado por RAM-r_{inicio}, RAM-$r_{inicio+1}$, ... RAM-r_{fim}. A segunda proposta se resume em aplicar uma função de espalhamento diretamente no identificador, ou seja, somente existirá "RAM" como identificador.

Na próxima seção, apresentamos nossa proposta para o problema de buscas por intervalo nas tabelas de *hash* distribuídas, que se baseia na combinação das duas abordagens acima mencionadas.

6.3.2 Estrutura Simplificada

A estrutura, que denominamos Lista para Busca por Intervalo (LBI), é uma lista distribuída, ordenada por um valor e que permite buscas simples e por intervalo. As buscas e inserções de itens, dado um identificador, utilizam

$O(\log n)$ trocas de mensagens, onde n corresponde à quantidade de itens armazenados na LBI. Esses itens são indexados por um valor e inseridos na posição correta da LBI de modo que a lista esteja sempre ordenada de forma crescente. Nos referiremos aos itens armazenados na tabela de *hash* distribuída como recursos.

Na Figura 6.4, mostramos a estrutura com oito recursos e identificadores não repetidos. Cada recurso armazenado tem seu respectivo identificador (definido como a união do nome do recurso e o valor) e ponteiros para seu predecessor, seu sucessor e uma tabela de ponteiros (*fingers*) para outros recursos, usada para acelerar as buscas.

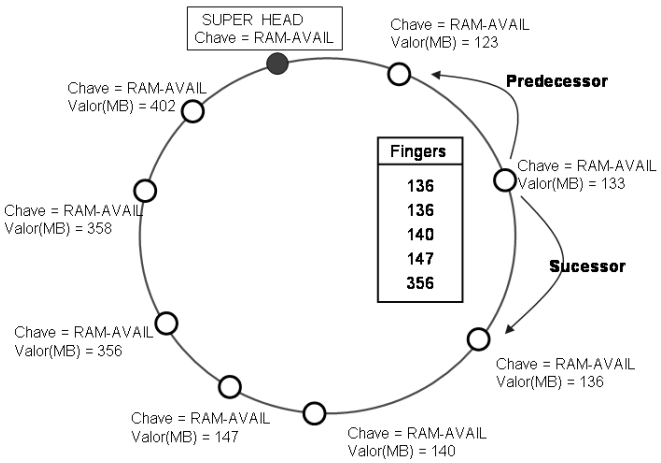


Figura 6.4: A estrutura Lista para Buscas por Intervalo sem valores repetidos.

A tabela de *fingers* de um recurso r é um conjunto de identificadores onde cada registro i dessa tabela corresponde ao recurso que sucede r com um valor maior ou igual a $r + 2^{i-1}$ (explicado na Seção 4.3.1). Finalmente, a LBI possui uma cabeça de lista chamada *super head* que mantém o nome do recurso armazenado, seu sucessor e a tabela de *fingers*. É usada nas buscas e inserções de recursos como veremos na Seção 6.3.4.

6.3.3 Estrutura Estendida

Em sistemas reais, é bastante comum encontrar situações em que a busca por um valor, dado um identificador, tenha como resposta vários recursos. Por exemplo, supondo que o identificador é a pessoa e o valor é a idade, deseja-se

buscar todas as pessoas com 17 anos de idade. Até o momento da escrita desta tese, nenhum outro trabalho [AX02, GS04, YB04, RHS03] parece solucionar este tipo de problema.

Como mostra a Figura 6.5, 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 denominaremos (*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 mostrado na Seção 6.3.4. Sem essa tabela, o retorno de todos os valores será de ordem linear.

Na lista para valores repetidos, os recursos não estão ordenados de forma alguma e somente é criada se o valor é repetido e é apagada se ela não contém mais recursos repetidos. Finalmente, a tabela de *repeated fingers* para um recurso r contém as entradas i 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 6.5, vemos que o recurso com valor 133 tem em sua tabela de *repeated fingers* o par com ID = D, que representa o recurso que está a uma distância (do recurso com valor 133) igual a 4 , ou seja 2^i com $i = 2$.

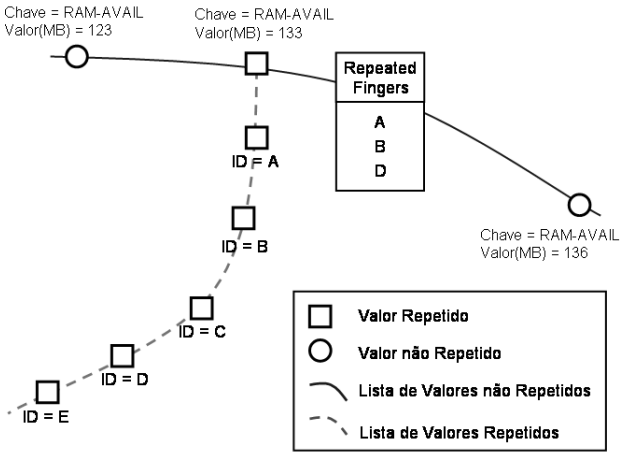


Figura 6.5: A estrutura Lista para Busca por Intervalo com valores repetidos.

6.3.4 Buscas por Intervalo

A Figura 6.6 mostra o algoritmo utilizado para buscar recursos pertencentes a um intervalo $[i, e]$. 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, chama-se 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. Esta quantidade de troca de mensagens é devida a iteração da linha 4, que envia a mensagem de busca para cada recurso da tabela de *fingers*. Esses recursos, por sua vez, repetem o processo de envio da mensagem, o que produz que a busca passe por todos os recursos do intervalo. Como conseqüência, a busca é assintoticamente linear ao número de recursos devolvidos.

```
1:  SEARCH(i, e, r)
2:  lista ← ∅;
3:  se (r == null)
4:     devolve lista;
5:  valores ← r.obter_valores_por_intervalo(i, e);
6:  lista ← valores
7:  para cada valor v em valores
8:     recurso ← v.recurso;
9:     lista ← lista ∪ SEARCH(v, e, recurso);
10: devolve lista;
```

Figura 6.6: Algoritmo usado para buscar os valores contidos em um intervalo.

O algoritmo começa verificando se o recurso r é *null* (Linha 3), significando isto que r é o final da LBI. Se for *null*, não é necessário continuar o algoritmo e devolve-se uma lista vazia (Linha 4). Depois de obter do recurso r os valores contidos no intervalo $[i, e]$ (Linha 5), ocorre uma iteração sobre esses valores para obter novos valores (Linhas 7-9). Finalmente, o método SEARCH devolve uma lista com todos os recursos encontrados (Linha 10). O método obter_valores_por_intervalo (Linha 5) devolve todos os valores no intervalo $[i, e]$, com seu respectivos recursos, mantidos nas tabelas *fingers* e *repeated fingers* de r.

6.3.5 Inserção de um Novo Recurso

A Figura 6.7 mostra o algoritmo para inserção de um recurso r, com valor v, na posição correta da LBI. Nesta posição, o sucessor de r será o recurso com o valor maior, mais próximo de v.

O algoritmo começa verificando se o valor v a inserir é repetido ou não (Linha 3). Se o valor não for repetido, procura-se pelo *super head* do identificador e se ele não existir é criada a lista de valores não repetidos (Linhas 4-7). Depois de procurar pelo melhor sucessor s para o valor a inserir (Linha 8), é necessário fazer com que o predecessor de s aponte para r (Linha 10) e fazer com que o ponteiro sucessor de r aponte para s (Linha 11). No caso do valor estar repetido (Linha 14), é somente necessário obter o recurso *topo* com esse valor (Linha 16) e inserir r no topo da lista ligada de valores repetidos (Linhas 17-19). A quantidade de troca de mensagens necessária para inserir o novo recurso é $O(\log n)$.

```
1:  PUT(identificador, recurso)
2:  v ← recurso.value;
3:  if (recurso.não_for_repetido()) then
4:     if (não existe super_head) then
5:        super_head.cria(identificador);
6:        ligue_super_head_a_recurso();
7:     end if
8:     sucessor ← super_head.obter_melhor_sucessor(v);
9:     if (existe sucessor) then
10:       ligue_predecessor_do_sucessor_a_recurso();
11:       ligue_sucessor_do_recurso_a_sucessor();
12:    else
13:       ligue_recurso_a_sucessor();
14:    end if
15: else
16:    topo ← obter_recurso_repetido(v)
17:    novo_topo ← recurso;
18:    ligue_novo_topo_a_topo();
19:    ligue_predecessor_do_topo_a_novo_topo();
20: end if
```

Figura 6.7: Algoritmo para adicionar um novo recurso.

O método ligue_recurso1_a_recurso2 faz com que o ponteiro sucessor do recurso 1 aponte para o recurso 2 e o predecessor do recurso 2 aponte para o recurso 1. O método obter_melhor_sucessor(valor_buscado) obtém o recurso com o valor mais próximo do *valor_buscado* entre todos os recursos da estrutura LBI e é realizado em $O(\log n)$ passos [SMK+03].

6.3.6 Algoritmo de Estabilização da Estrutura

Como vimos na seção anterior, o novo recurso, antes de ser inserido, deve procurar pela posição correta na LBI (*sucessor*), pois a LBI é uma lista ordenada. Quando finalizada a inserção, o predecessor desse *sucessor* não aponta para o novo recurso, nem o novo recurso aponta para o predecessor. Portanto, depois de executado este algoritmo, o sucessor e o predecessor de um recurso pertencente à LBI apontarão para o recurso certo (chamaremos isso de estabilização). O exemplo a seguir mostra porque um recurso precisa de uma estabilização.

A Figura 6.9(a) é o estado final atingido depois de inserir os recursos p, s e r (nessa ordem) com o algoritmo apresentado na Figura 6.7. Na Figura 6.9(a), podemos observar que o ponteiro predecessor do recurso s está apontando para p (sendo que deveria apontar para r) portanto, a estabilização se faz necessária cada vez que um recurso r é inserido no meio da estrutura LBI. A estabilização da estrutura é realizada em tempo constante e é executada periodicamente sem a intervenção dos usuários.

A Figura 6.8 mostra o algoritmo para a estabilização que está baseado no trabalho de Stoica [SMK+03]. Uma vez obtido o predecessor p e o sucessor s de um recurso r, seus ponteiros são atualizados.

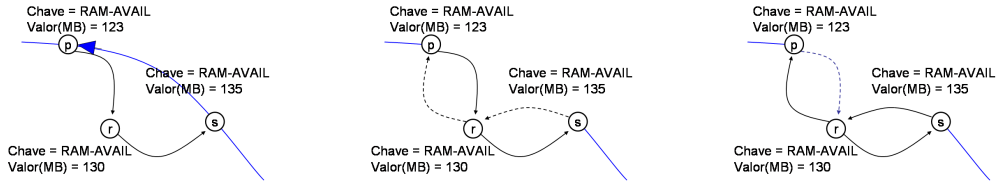
Suponha que o recurso r fosse adicionado à LBI como mostrado na Figura 6.9(a). Quando o método ESTABILIZA é chamado pelo recurso r, a execução das Linhas 6 e 7 são mostradas nas ligações da Figura 6.9(b). Finalmente, quando o método ESTABILIZA é chamado por p, a execução da linha 5 é mostrada na Figura 6.9(c) e a estrutura se estabiliza.

```
1: ESTABILIZA()
2: recurso ← obtem_recurso_aleatório();
3: sucessor ← recurso.sucessor;
4: predecessor ← sucessor.predecessor;
5: atualiza_sucessor_do_recurso();
6: atualiza_predecessor_do_recurso();
7: atualiza_predecessor_do_sucessor();
```

Figura 6.8: Algoritmo que atualiza o predecessor e o sucessor de um recurso.

6.3.7 Algortimo de Estabilização da Tabela de *Repeated Finger*

Esse algoritmo tem por objetivo atualizar a tabela *repeated fingers* de um recurso r e somente é executado quando r possui um valor repetido. Cada entrada i desta



(a) O recurso r foi adicionado entre p e s

(b) Linhas 6 e 7 da Figura 6.8 quando o método Estabiliza é chamado por r

(c) Linha 5 da Figura 6.8 quando o método Estabiliza é chamado por p

Figura 6.9: O método Estabiliza aplicado a diferentes recursos.

tabela, quando finalizado este processo, terá o recurso que está a uma distância de 2^i, de r, na lista de valores repetidos.

O algoritmo desse processo é mostrado na Figura 6.10. A Linha 2 se encarrega de obter um recurso r. Existem diferentes formas de fazer isso, mas em nossa implementação preferimos obter, de forma aleatória, qualquer recurso pertencente à LBI e que está armazenado no par que está executando o processo. Se r é um valor repetido (Linha 3) devemos obter o sucessor que está a uma distância *distancia_procurada* de r (Linha 6) e notificar r sobre o sucessor encontrado (Linha 8).

```
1:  ESTABILIZA_REPETIDOS()
2:  recurso ← obtem_recurso_aleatório();
3:  if recurso.for_repetido() then
4:      distancia_procurada =
5:          obtem_distancia_aleatória();
6:      sucessor =
7:          encontra_sucessor_pela_posição(distancia_procurada);
8:      recurso.notifique_recurso_repetido(
9:          distancia_procurada, sucessor);
10: end if
```

Figura 6.10: Algoritmo para estabilizar os valores repetidos.

Para entender melhor a notação do código: O método *obtem_distancia_aleatoria* obtém um valor aleatório potência de 2. O método *encontra_sucessor_pela_posiao* (Linha 7) devolve o identificador do recurso p que está associado à entrada i da ta-

bela de *repeated fingers*, onde $i = \log(distancia\_procurada) - 1$. A partir de p, a $distancia\_procurada$ diminui em 2^i e o processo se repete de maneira recursiva até que a distância seja zero. O método $notifique\_recurso\_repetido$ (Linha 8) atualiza a tabela de *repeated fingers* de um recurso com o *sucessor* encontrado.

Capítulo 7

Trabalhos Relacionados

Este capítulo apresenta alguns dos trabalhos relacionados aos protocolos descritos nas Seções 5 e 6. Deve-se salientar que os sistemas descritos representam apenas uma pequena fração dos sistemas existentes.

7.1 Trabalhos Relacionados à Interligação

Com o surgimento da Computação em Grade, diferentes sistemas foram desenvolvidos pela comunidade acadêmica e pela indústria. Esta seção analisa alguns dos trabalhos representativos na descoberta e interconexão entre diferentes aglomerados de uma grade.

7.1.1 Condor

O sistema Condor [Con, LLM88] é um dos sistemas de grade mais antigos. Desenvolvido em 1988 pela Universidade de Wisconsin, provê uma arquitetura para realizar tarefas que necessitam um alto poder computacional usando os recursos ociosos dos computadores. Existem diferenças em relação ao InteGrade que são analisadas na dissertação de mestrado de Andrei Goldchleger [Gol04].

A arquitetura do Condor é formada por um aglomerado de computadores, chamado *Condor Pool*, que é monitorado pelo Administrador Central. O Administrador Central é o encarregado de coletar e distribuir as tarefas entre os computadores do aglomerado.

Para compartilhar recursos entre os diferentes aglomerados (o que é chamado de *Flocking*), Condor inclui mecanismos de descoberta de outros aglomerados baseado em configurações estáticas feitas manualmente. O processo ocorre da seguinte forma: cada administrador central precisa conhecer outros administradores centrais aos quais poderá enviar uma tarefa para ser executada. Devido ao fato destes admi-

nistradores mudarem suas configurações constantemente em sua disponibilidade, seja de capacidade ou de endereço, este mecanismo limita a escalabilidade do Condor.

Para melhorar a escalabilidade no processo de descoberta, os pesquisadores do Condor apresentaram em 2003 um protótipo de sua nova arquitetura de organização de aglomerados baseadas em uma estrutura *Par-a-Par* [BZH03]. Nessa arquitetura, Condor usa a Tabela de *Hash* Distribuída baseada em Pastry[1] para descobrir diferentes aglomerados Condor. Aqui, cada Administrador Central é registrado na estrutura com um identificador único.

Com esta nova arquitetura, Condor propõe três alternativas para obter os aglomerados aos quais pedir a execução de uma tarefa.

1. A primeira alternativa consiste em que o Administrador Central envie, através de *broadcast*, uma mensagem de pedido de execução de uma tarefa a todos os outros Administradores Centrais pertencentes à estrutura *Par-a-Par* formada pelos aglomerados do Condor.

2. A segunda alternativa consiste em usar a camada *Par-a-Par* para a localização eficiente dos outros aglomerados. Nessa camada, usarão especificamente os identificadores dos administradores registrados na tabela de roteamento do Pastry, que garante uma certa proximidade física entre os Administradores Centrais. Com isso, as tarefas executadas e as mensagens trocadas entre os administradores não percorrerão longas distâncias.

3. A terceira alternativa é usar uma mensagem de requisição para buscar novos aglomerados e que será enviada a alguns dos Administradores Centrais registrados na estrutura. Essa mensagem será propagada por eles com um tempo de vida que irá se decrementando em cada propagação.

A primeira alternativa tem um sério risco de escalabilidade devido à sobrecarga na rede pela quantidade de mensagens trocadas. Devemos considerar também que se todos os administradores enviam uma mensagem de pedido a um só administrador, este ficará sobrecarregado.

Um dos problemas de utilizar a tabela de roteamento do Pastry, como apresentado na segunda alternativa, é que os Administradores registrados nessa tabela estão próximos em termos das chaves armazenadas e não em termos de latência. Com isso podem existir outros Administradores que estejam mais próximos do que os obtidos.

No processo de descoberta de novos aglomerados, existem diferenças do Condor em respeito ao protocolo apresentado no Capítulo 5. Nosso protocolo utiliza

[1] Ver sobre o Pastry na Seção 4.3.2

uma modificação da segunda e da terceira alternativa. No caso da segunda alternativa, utiliza-se a camada de rede (e não a tabela de roteamento) para obter os aglomerados. No caso da terceira alternativa, nosso protocolo possui um repositório (*vide* Seção 5.1.2) que contém os aglomerados descobertos e que não necessariamente pertencem à tabela de roteamento. É importante destacar que este repositório pode conter aglomerados que estejam mais próximos em termos de latência que os da tabela de roteamento citada anteriormente.

7.1.2 Globus

Globus [Glo] é um sistema de Grade que tem o objetivo de ajudar na resolução de tarefas que requerem grande poder computacional usando os recursos de computadores distribuídos em redes de grande área [FK97]. Atualmente é o projeto de maior impacto na área de Computação em Grade que envolve diversas instituições de pesquisa e grandes empresas tais como a IBM e a Microsoft.

A arquitetura do Globus apresenta um conjunto de serviços que servem como uma infra-estrutura base para o desenvolvimento de aplicações de grade. Com esse fim, ele concentra-se principalmente em duas tarefas: a primeira corresponde ao desenvolvimento de mecanismos de baixo nível (comunicação, autenticação, etc.) que podem ser usados para implementar serviços de alto nível e a segunda, a técnicas que permitem que serviços observem e administrem as operações dos mecanismos de baixo nível (interfaces de programação paralela, escalonadores, etc.).

Dentro desses serviços, o Globus disponibiliza o serviço de descoberta e monitoramento (*Monitoring and Discovery System*), que é o responsável por reunir e monitorar os recursos disponíveis na Grade. A arquitetura do serviço de descoberta consiste em dois elementos básicos [CFFK01]:

- Uma grande e distribuída coleção de provedores de informação, *Information Providers*, que permitem o acesso a informações sobre uma determinada entidade como, por exemplo, recursos de um computador, tipo de rede utilizado, capacidade de transferência, etc. A informação é estruturada em termos de um modelo de dados tomado do LDAP [ZS04] onde cada entidade é descrita por um conjunto de objetos de tipo (atributo, valor).
- Serviços de alto nível, coletores, administradores, índices e entidades que respondem a requisições feitas pelos provedores de informação. Em particular, essas entidades chamadas de "diretório de agregado" (*Aggregate Directories*) facilitam a descoberta e monitoração de recursos.

O protocolo do serviço de descoberta permite que um provedor de informação possa se registrar a um ou mais "diretórios de agregado" aos quais poderá fazer

requisições e os diretórios se encarregam de se comunicar com outros diretórios. Então, quando um recurso quer ser disponibilizado na Grade deve se comunicar com um provedor de informação que será o responsável por esse recurso. Por sua vez, esse provedor de informação se registrará em um (ou vários) "diretório de agregado".

A descoberta dos recursos de uma Grade começa quando o provedor de informação solicita, aos seus diretórios conhecidos, algum recurso. Os diretórios consultam em sua base de dados, implementado utilizando LDAP, se esse recurso é conhecido e o fornece caso o tenha. Caso não o tenha, o diretório pode propagar a consulta a outros diretórios.

Uma das restrições da arquitetura baseada em provedores de informação e diretórios de agregado é que a configuração dos serviços de informação pressupõe que os provedores conhecem os endereços dos diretórios aos quais devem se registrar. O Globus no momento depende de uma configuração manual, na qual os usuários e administradores do sistema configuram os provedores de informação com os endereços dos diretórios de forma estática. Essa abordagem tem sérios problemas de escalabilidade, mas é de fácil utilização no caso de poucos diretórios de agregados.

Existem diferenças do Globus em respeito ao protocolo apresentado no Capítulo 5. Primeiro, nosso protocolo utiliza uma estrutura *Par-a-Par* que permite uma melhor escalabilidade no processo de descoberta. A melhor escalabilidade é dada devido a não dependência de servidores centrais ("diretórios de agregado") para obter novos aglomerados. Nosso protocolo utiliza qualquer par para obtê-los, através da propagação de uma mensagem como vimos na Seção 5.2.3. Segundo, a descoberta de novos aglomerados, pelo Globus, não prevê a latência entre eles, portanto, pode ocorrer de o aglomerado descoberto não ser a melhor solução para uma comunicação eficiente, como vimos na Seção 5.

7.1.3 Gridbus

O Gridbus [gri], desenvolvido pelo departamento de Ciência da Computação da Universidade de Melbourne, é um projeto voltado à criação de especificações, arquiteturas e serviços orientados à construção de aplicações científicas (*eScience*) e de comércio (*eBusiness*) que precisam de um alto poder computacional. O objetivo principal do Gridbus é a de regular a demanda por recursos da Grade, provendo incentivos econômicos para os provedores desses recursos. A idéia dos incentivos econômicos visa atenuar o problema dos usuários que somente consomem mas não compartilham recursos [BV04].

A arquitetura do Gridbus atualmente é composta por computadores que compartilham recursos, independentemente de se eles pertencem a um aglomerado, a uma organização virtual, *Virtual Organization*, ou a uma empresa virtual *Virtual*

Enterprise [YVB03]. Esses recursos são registrados como provedores de serviços da Grade (*Grid Service Providers*) em um diretório chamado de *Grid Market Directory* (GMD), o qual permite a descoberta desses recursos para as máquinas pertencentes à Grade. Este modelo é muito parecido com o Sistema de Descoberta e Monitoração do Globus.

O processo de descoberta de recursos funciona da seguinte forma: quando um cliente (*Grid Service Consumer*) precisa de um recurso para a execução de uma tarefa, ele deve contatar o diretório onde estão registrados os recursos. Este, por sua vez, informará os recursos disponíveis na Grade e que poderão ser utilizados pelo cliente. É importante destacar aqui que os endereços dos diretórios devem ser conhecidos, o que gera um problema de escalabilidade no caso de existirem muitos diretórios.

Para melhorar o problema da descoberta de novos diretórios e recursos, Gridbus apresentou no final do 2004 uma arquitetura diferente. Neste modelo, chamado de Federação de Grades (Grid-Federation), a Grade consiste em aglomerados de computadores que estão unidos através de uma estrutura *Par-a-Par* o que permite a descentralização dos GMD da arquitetura anterior. Cada aglomerado terá como responsável um agente (*Grid Federation Agent*) que dissemina a descrição das características do cluster através da rede usando para isso o protocolo do Chord[2] para o envio de mensagens e armazenamento de recursos.

O agente, além de disseminar a informação do aglomerado, permite a cooperação entre diferentes aglomerados, utilizando a camada *Par-a-Par* para a descoberta e registro de recursos da Grade. Assim, quando um usuário deseja executar uma tarefa, este agente verifica se esta pode ser executada localmente. Senão, através da federação, a tarefa pode ser transferida a um outro agente que satisfaz os requisitos da tarefa.

A busca por outros agentes é baseada na propagação da descrição de um aglomerado qualquer (*quote*) que é enviada a todos os agentes da Grade. Por exemplo, quando um agente entra na Grade e se registra na estrutura *Par-a-Par*, é enviada uma mensagem de *broadcast* a todos os outros agentes registrados na Grade. Com isso, um agente conhecerá os outros aglomerados e poderá enviar tarefas para serem executadas nestes, de acordo com o critério de busca, seja rapidez na comunicação, melhores máquinas, etc. Uma desvantagem desta proposta é que enviar as características a todos os outros agentes pode ser pouco escalável no caso de haver muitos agentes.

No processo de descoberta de novos aglomerados, existem várias diferenças do Gridbus em respeito ao protocolo apresentado no Capítulo 5. Entre elas: (1)

[2] Ver sobre o Chord na Seção 4.3.1.

nosso protocolo não faz *broadcast* das informações a todos os pares da estrutura (que afeta diretamente a escalabilidade) e (2) os aglomerados descobertos têm uma proximidade em termos de latência, que pode melhorar a rapidez da comunicação entre eles, como vimos na Seção 5.1.

7.2 Trabalhos Relacionados à Localização

Esta seção apresenta alguns trabalhos relacionados a localização de recursos dado um intervalo de valores, a maioria usando a Tabela de *Hash* Distribuída para o armazenamento e busca destes.

7.2.1 *Prefix Hash Tree*

Uma das primeiras alternativas para buscar recursos por intervalos de valores nas tabelas de *hash* distribuídas foi desenvolvida conceitualmente por Ratnasamy et al [RHS03]. Nesse trabalho, é proposta a estrutura de árvore *Prefix Hash Tree* (ou PHT) na qual cada nó intermediário da árvore tem associado uma etiqueta que corresponde a um prefixo do domínio indexado e as folhas da árvore tem os dados do domínio. Esses prefixos estão armazenados na tabela de *hash* distribuída como $(chave, valor) = (etiqueta, n)$ e podem ser obtidos usando as funções de recuperação de chaves próprias das implementações das tabelas de *hash* distribuídas.

Para entender a estrutura PHT, podemos compará-la com um *trie*[3], geralmente utilizado para indexar palavras sobre um alfabeto. As folhas (ou seja as palavras completas) ficam penduradas em um nó intermediário que tem como etiqueta um prefixo comum a todas essas palavras. Por exemplo, as palavras cantar, canto, cânticos seriam folhas penduradas a um nó intermediário com etiqueta "cant".

A busca por um intervalo de valores na PHT pode ser feito de duas formas: (1) começa-se perguntando à tabela de *hash* distribuída pelo nó raiz da PHT que contém a chave do dado indexado (por exemplo RAM-Disponível). Esse nó devolve os nós pendurados que correspondem ao prefixo do intervalo procurado. Este processo é recursivo até chegar as folhas onde estão os dados que serão devolvidos. (2) Para cada valor do intervalo procurado lhe é extraído seu prefixo. No caso desse prefixo existir na tabela de *hash* distribuída, esta devolve o nó responsável por esse prefixo e a busca para localizar as folhas começa desse nó. Isso melhora o desempenho comparado com a primeira alternativa porque não tem que se percorrer a árvore desde a raiz para localizar os dados.

[3] *Do termo re**trie**val, o trie é uma estrutura eficiente para armazenamento e buscas de palavras com prefixo comum. Nesta estrutura, as palavras são armazenadas nas folhas [dlB59].*

A diferença da nossa proposta é que a estrutura LBI não precisa de uma estrutura extra (o *trie*) para ordenar os valores dos recursos, porque a ordem é dada pela própria LBI. Além disso, as árvores apresentam um problema extra de manutenção. Por exemplo, se um filho de um nó sair da PHT, a estrutura tem que se adequar a esta saída organizando os nós intermediários de "todos" os prefixos do nó que saiu. Em nosso caso, somente as tabelas de *finger*, o sucessor e o predecessor de um nó tem que ser atualizadas.

7.2.2 *Extended PHT*

No mesmo contexto das árvores, Gao et al. [GS04] utilizam a PHT para indexar valores. Existem duas diferenças com respeito ao modelo apresentado na seção anterior. Primeiro, nesta extensão são utilizados os nós intermediários da PHT para armazenar os recursos. Com isso, para localizar os dados de um intervalo não é necessário chegar até as folhas. Segundo, cada nó desta árvore tem um conjunto de réplicas que evita que os nós fiquem sobrecarregados. Assim, mesmo que as buscas comecem pela raiz da estrutura, as réplicas mantém o desempenho distribuindo as consultas entre elas.

Para a busca dos recursos contidos em um intervalo utilizando a PHT, os autores explicam três alternativas:

- Procurar na tabela de *hash* distribuída pelo identificador do dado armazenado, ou seja a raiz (por exemplo RAM-disponível). Esse nó terá como filhos todos os recursos e será responsável por devolver os dados armazenados nos filhos. Assim, a busca por qualquer intervalo sempre será constante pois a árvore tem somente um nível.

  O problema desta alternativa é que podem existir diferentes valores que compartilhem o mesmo identificador [RLS+03], e mesmo que a função de *hash* forneça uma distribuição proporcional dos nós que armazenam os identificadores [KR04], o nó responsável pelo único identificador deverá ter problemas de sobrecarga. Para entender isso, vamos supor que o dado a ser indexado é RAM e que essa chave será armazenada em um nó x. Todos as máquinas que desejem ter sua RAM indexada ou que procurem por um intervalo da chave terão que acessar o nó x produzindo uma queda no desempenho da entrega das respostas devido à sobrecarga nele.

- Procurar na tabela de *hash* distribuída por cada valor do intervalo buscado. Se esse valor existir na tabela então é devolvido o recurso armazenado.

  Esta alternativa traz alguns problemas: (1) a busca terá que ser dividida nos valores do intervalo, ou seja serão realizadas $f - i + 1$ buscas. No caso do

intervalo ser muito grande ou o intervalo ter poucos dados armazenados, este tipo de busca é pouco escalável e pouco eficiente respectivamente. (2) Se o intervalo contém valores contínuos (não discretos) a divisão do intervalo não pode ser definida.

- A proposta dos autores é uma combinação das duas alternativas mostradas acima. Com respeito à primeira alternativa, cada nó da estrutura tem associado réplicas que evitam a sobrecarga nas consultas. No caso da segunda alternativa, a busca é dividida nos valores do intervalo mas o nó responsável tem associado filhos aos quais a busca será propagada. Nessa propagação são encontrados os dados (sem ter que chegar às folhas) o que melhora o desempenho na devolução dos valores do intervalo.

7.2.3 *Skip Graph*

O trabalho que mais se assemelha com à estrutura apresentada na Seção 6.3, mas que não utiliza as tabelas de *hash* distribuídas para o armazenamento dos recursos, é o *Skip Graph* [AS03]. *Skip Graph* é uma estrutura *Par-a-Par* que estende a estrutura Skip List [Pug89] provendo tolerância a falhas e replicação de dados.

Para entender o *Skip Graph* devemos analisar antes o *Skip List*. *Skip List* é uma estrutura de árvore balanceada e organizada como uma torre de listas esparsas encadeadas. Como mostrado na Figura 7.1, o nível zero da *Skip List* contém todos os valores em uma ordem crescente dada pela chave indexada. Um valor de um nível i aparece no nível $i+1$ dada uma probabilidade, ou seja, os níveis de cima são formados com recursos dos níveis de baixo que têm uma certa probabilidade de aparecer. Cada recurso de um nível do *Skip List* tem ponteiros a recursos posteriores na lista. O último nível do *Skip List*, muito parecido com a nossa estrutura, é formado por ponteiros que, no pior caso, podem resultar em uma busca linear.

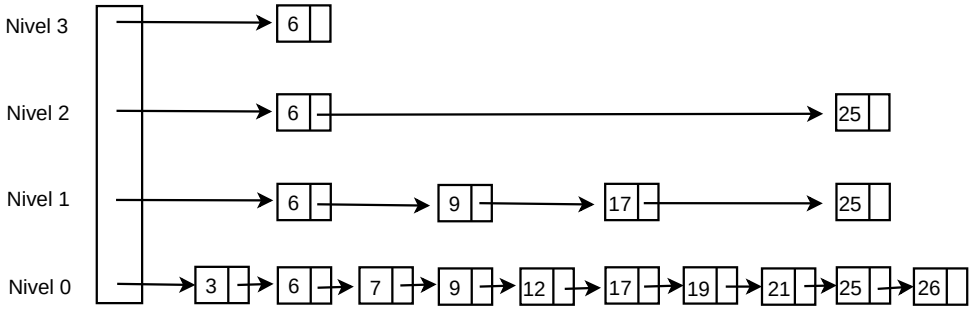


Figura 7.1: A estrutura *Skip List*.

A idéia do *Skip Graph* é manter uma coleção de *Skip Lists* onde vão ser compartilhados os primeiros níveis. Com isso, cada nível i vai ter mais recursos onde efetuar as buscas (devido à união dos níveis i da coleção de *Skip List*) produzindo duas melhorias: (1) a probabilidade de que um recurso fique sobrecarregado é menor (2) a falha de um recurso, por exemplo o nó raiz do *Skip List*, não afetará a estrutura.

Para buscas por intervalo neste tipo de estrutura se procede da seguinte maneira. Dado um intervalo $[i, f]$ começa se perguntando aos recursos raiz do *Skip Graph* (note que existem tantos recursos quantos *Skip List* tenha o *Skip Graph*) se o valor deles estão dentro do intervalo procurado. Os recursos raiz devolvidos conterão ponteiros a outros recursos. Para para cada um desses recursos apontados, continua se repetindo o processo até achar todos os valores. Com isto, o processo faz $O(m \log n)$ troca de mensagens.

Como dito anteriormente, nós não temos uma estrutura adicional para ordenar os valores dos recursos e os ponteiros da tabela de *fingers* permitem realizar uma busca em tempo logarítmico.

Capítulo 8

Simulação

Há inúmeros trabalhos que buscam investigar e desenvolver novas técnicas para construir serviços de Internet escaláveis e confiáveis, incluindo nisso redes *Par-a-Par*, redes sobrepostas (*overlay*), replicação em redes de grande área, etc. Esses sistemas são projetados para uma rede formada por um grande numero de nós distribuídos na Internet.

Para testar e avaliar tais sistemas, os desenvolvedores devem implantá-los em cenários realistas, como por exemplo em redes grandes, estruturadas ou ad-hoc. No entanto, é muito difícil implantar e administrar esses sistemas em nós espalhados em diversos sítios na Internet. Assim, os resultados obtidos dessa implantação, na prática, não são nem reprodutíveis nem previsíveis porque as condições e comportamentos mudam rapidamente e não estão sujeitos ao controle nem manipulação do pesquisador [VYW+02].

No caso de protocolos, um dos objetivos primordiais no seu projeto tem sido a robustez. Desde a era pré-Internet, a maioria dos esforços iniciais nesse aspecto foram os modelos a prova de falhas, onde a falha dos nós eram total e facilmente detectáveis por outros nós e podia ser corrigida. O protocolo IP, por exemplo, é robusto contra vários tipos de falhas usando pouquíssimos recursos.

As dificuldades de experimentação com redes reais de grande porte fazem com que a simulação tenha um papel importante na descoberta de possíveis problemas que o protocolo possa ter.

8.1 Escolha e Implantação

A simulação do protocolo proposto em nosso trabalho terá o objetivo de verificar se ele atende a requisitos importantes de desenvolvimento deste tipo específico de aplicação, tais como: escalabilidade, baixo tempo de resposta, pequena quantidade de mensagens trocadas etc. Para isso, o primeiro passo foi escolher uma ferramenta

de simulação de redes de grande envergadura que permita analisar o protocolo e obter resultados dos tipos supracitados.

Para isso, foi escolhido o simulador do Bamboo [Bam], que foi desenvolvido com o objetivo de poder avaliar as aplicações que usam a tabela de *hash* distribuída Bamboo. É importante destacar que foram feitas algumas adaptações no código do simulador para obter dados da simulação que serão analisados na Seção 8.2. As adaptações foram feitas de acordo com as recomendações dadas pelo desenvolvedor do Bamboo obtidas através das interações feitas durante o desenvolvimento dos protocolos. A primeira adaptação foi a de permitir a tabela de *hash* recuperar um recurso, dada somente a chave de indexação, como "RAM" ou "CPU" (antes era necessário utilizar outros parâmetros, como a data em que foi criado, tempo de expiração, etc.). A segunda adaptação foi a de utilizar estruturas que armazenassem a quantidade de bytes enviados e recebidos pelos pares pertencentes à tabela de *hash* distribuída na troca de mensagens entre eles (ou seja, no ingresso de um par e nas atualizações mostradas na Seção 5.2).

Os passos necessários para simular as aplicações e protocolos no simulador do Bamboo são:

1. O usuário especifica a topologia onde será testada a aplicação.
2. O usuário especifica os caminhos entre dois pares quaisquer da rede com sua respectiva latência.
3. A aplicação ou protocolo a testar é instanciado pelo simulador.

No primeiro passo, utilizamos diferentes estruturas de rede, algumas criadas para casos específicos (como pares que estivessem muito perto uns dos outros e existissem poucos roteadores entre eles) e outras já criadas por outros desenvolvedores para testes.

No segundo passo, tivemos que colocar valores (que representaram a latência) aos caminhos entre dois pares. Neste passo, testamos com diferentes valores (a maioria eram dados de forma aleatória) nas estruturas criadas por nos. Nas estruturas criadas por outros desenvolvedores, algumas já vinham com valores próprios.

No terceiro passo, é executado o simulador, onde se especifica a quantidade x de pares que terá a simulação, junto com a estrutura da rede e seus caminhos. O simulador cria então x instâncias de pares em uma máquina, mas simula a troca de mensagens como se estivessem em diferentes máquinas.

8.2 Resultados Experimentais da Intercomunicação

Nesta seção, avaliamos o Protocolo de interligação, apresentado na Seção 5, usando o simulador do Bamboo que permite configurar o número de nós que terá a simulação. Nossa plataforma para simular o protocolo foi um computador com 1 GB de RAM e sistema operacional GNU/Linux – Fedora Core 2. A seguir, mostraremos o modelo que usamos para simular o protocolo e os diferentes tipos de medições que foram feitas para verificar se o protocolo é escalável.

8.2.1 Ambiente de Teste

Nosso objetivo foi simular o protocolo para que funcionasse como se cada par fosse um GRM do InteGrade. Usamos um modelo de rede com as seguintes características. Existem dois tipos de nós, os roteadores e os clientes. Os nós clientes somente podem se unir aos nós roteadores e um nó roteador deve estar unido a outros roteadores. Finalmente, 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.

8.2.2 Custo de Ingresso na Rede

A medida do tempo levado por um par para entrar na rede, em função da quantidade de pares que já ingressaram é muito importante para verificar a escalabilidade. Neste experimento usamos de 47 até 750 nós clientes (simulando GRMs do InteGrade) e 300 nós roteadores. A simulação foi feita da seguinte maneira: a estrutura de rede da simulação estava composta por 47, 100, 300, 500 e 750 pares, os quais eram adicionados um a um. Nos últimos 10 ingressos foram medidos o tempo (em segundos) que o par ingressante demorou em encontrar a referência a um outro par e o tempo (em segundos) que demorou em encontrar um par mais próximo. Finalmente foi calculada a média de cada medida.

A Figura 8.1 mostra que o custo de entrar na rede se mantém quase constante, independente da quantidade de pares que já tenham ingressado. Como vimos na Seção 5.2.1, para ingressar na rede o par só depende de uma conexão lógica (referência) com um outro. Para obter essa referência, deve primeiro obter alguns pares (podemos considerar esse processo como constante) onde se conectar e depois obter a menor latência. Supondo que a quantidade de pares obtidos é relativamente pequena, o processo para obter a menor latência (usando o método proposto no ponto 2 da Seção 5.2.1) também pode ser considerado constante. É importante destacar, que a obtenção e escolha de um par (Linha 2 e 3 da Figura 5.3) só demora

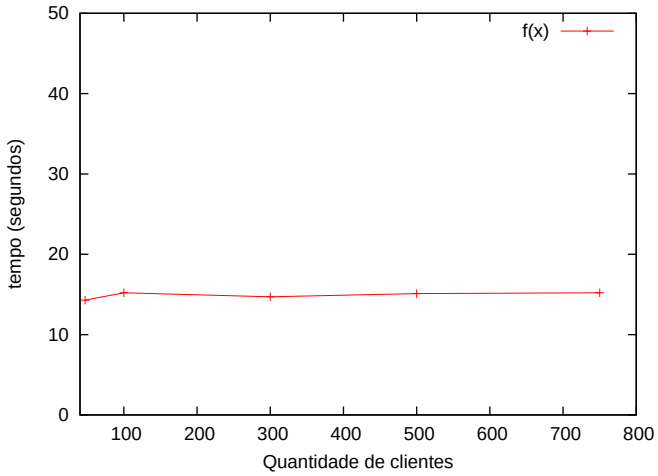


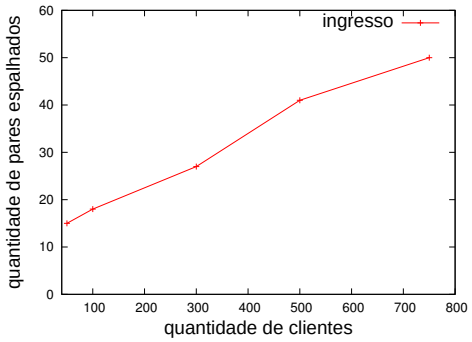
Figura 8.1: Custo em segundos da entrada de um par na rede.

3 segundos e que os outros 10 segundos são gastos em buscar um par mais próximo (Linhas 4-13 da Figura 5.3).

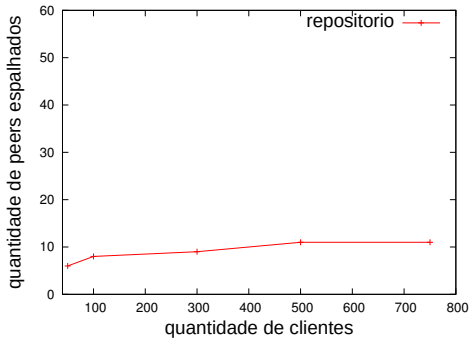
8.2.3 Quantidade de pares espalhados

Quando um par ingressa na rede, o Protocolo de Interligação se encarrega de lhe dar vários pares aos quais poder se conectar. Como esses pares são obtidos de forma aleatória, existe uma probabilidade de que os pares da vizinhança estejam distantes em termos de latência. Definimos então, como par espalhado, o par onde a maioria da sua vizinhança são referências a pares distantes, em termos de latência. O Protocolo de Interligação, apresentado na Seção 5.2, mostrou que a vizinhança de um par pode ser atualizada, tanto no processo de entrada de um par na rede, como na atualização do repositório. Neste contexto, temos que analisar essas duas possibilidades.

A simulação foi feita da seguinte maneira: a estrutura inicial da rede estava composta por 47 pares e dai se agregou outros pares até completar os 750. Nesta simulação, foi medido quantos pares são espalhados respeito ao total de pares pertencentes à rede. A Figura 8.2(a) mostra que no processo de entrada existem muitos pares espalhados, pelo fato do par escolhido para se conectar estar distante em termos de latência. Como mostrado na Figura 8.2(b), essa quantidade diminui quando o processo de atualização do repositório é executado pois neste processo tenta-se escolher um par que esteja mais perto e por conseguinte obter uma vizinhança de pares que também estejam mais perto (ver Seção 5.2.3). Por exemplo, podemos observar que quando um par entra em uma rede com 500 nós, 40 deles são espalhados, mas



(a) Quantidade de pares espalhados no processo de entrada dos pares na rede



(b) Quantidade de pares espalhados quando executado o processo de Atualização do Repositório

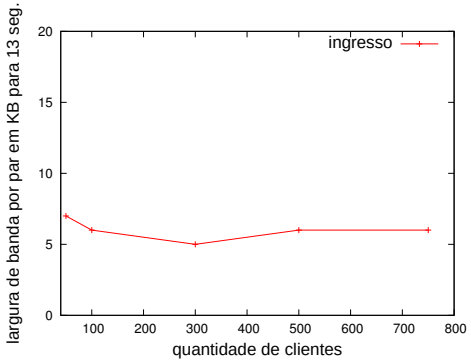
Figura 8.2: Quantidade de pares espalhados nos processos de entrada de um par e de atualização do repositório.

diminui a 10 quando executado o processo de atualização do repositório.

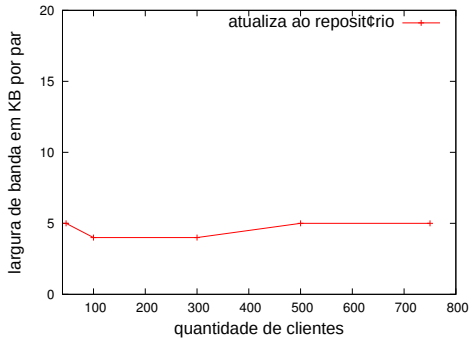
8.2.4 Largura de Banda Usada

Definimos a largura de banda usada como a quantidade de bytes que um par envia pela rede em um certo momento. O consumo da largura de banda usada por uma aplicação é um tema muito importante em recentes pesquisas que testam a escalabilidade dos protocolos *Par-a-Par*. Podemos observar na Figura 8.3(a) que o consumo de largura de banda ***por par***, quando se está executando o processo de ingresso de um par na rede, que demora 13 segundos, é quase constante. Isso devido ao fato de que, quando um par entra na rede, é necessário: (1) escolher o par de menor latência, dado uma quantidade constante de pares; (2) enviar uma quantidade de mensagens igual à quantidade de roteadores entre ele e o escolhido. Essas quantidades de bytes enviados pela rede (*ping* e *traceroute*) é praticamente constante.

O consumo da largura de banda feito pela atualização do repositório, mostrado na Figura 8.3(b), também é constante. Como apresentado na Seção 5.2.3, isso se deve ao fato de o repositório fazer requisições somente a uma quantidade limitada de vizinhos. Com isso, a quantidade de mensagens trocadas entre pares é constante.



(a) Largura de banda usada no processo de entrada de um par na rede



(b) Largura de banda usada no processo de Atualização do Repositório

Figura 8.3: Consumo de largura de banda usada nos processos de entrada de um par na rede e de atualização do repositório.

8.3 Resultados Experimentais da Localização

Nesta seção, avaliaremos o Protocolo de Localização apresentado na Seção 6 através de uma simulação. Nestes experimentos, o dado utilizado será a quantidade de memória RAM de uma máquina e o intervalo de valores estará definido por todos os números inteiros positivos possíveis para esse recurso. Para realizar os experimentos, nosso protocolo foi implementado em Java, usando Bamboo [Bam] como a implementação da tabela de *hash* distribuída e o seu simulador.

8.3.1 Ambiente de Teste

A simulação foi executada em um PC de 2.4 GHz, 2 GByte de memória RAM e sistema operacional GNU/Linux. O simulador foi adaptado para estimar, a partir da troca de mensagens, a largura de banda usada pelos pares.

8.3.2 Custo de Inserir um Recurso

A estrutura LBI é modificada cada vez que um recurso é inserido. O processo de inserir um recurso é um dos mais importantes na LBI pois reflete diretamente em quantos recursos será executado o processo de Estabilização (ou seja, a atualização do predecessor e o sucessor de um recurso), como mostrado na Seção 6.3.6.

Neste experimento, adicionamos 1000 recursos com valores incrementais consecutivos, com o objetivo de obter o pior caso de custo de registro, onde cada recurso

será adicionado ao final da LBI[1] e com um intervalo de tempo entre cada inserção. Neste contexto, temos duas alternativas a analisar: (1) É utilizada a tabela de *fingers* na busca da posição onde inserir o recurso. (2) A tabela de *fingers* não é utilizada.

A Figura 8.4 mostra o efeito de se usar ou não usar a tabela de *fingers*. Cada ponto do gráfico representa a quantidade de recursos aos quais será necessário aplicar o processo de Estabilização.

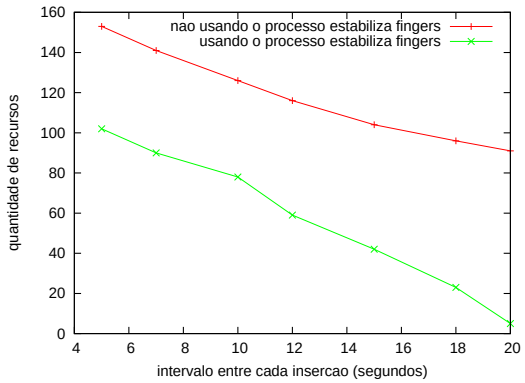


Figura 8.4: Custo de inserir um recurso usando ou não a tabela de *fingers*.

Podemos observar que a quantidade de recursos a serem estabilizados diminui quando: (1) o tempo entre cada inserção aumenta, (2) é utilizada a tabela de *fingers*.

Em (1), a mensagem que procura a posição final onde o recurso tem que ser adicionado não é influenciada pelos efeitos de sobrecarga de um par que pode estar processando uma outra mensagem. Em (2), a quantidade de recursos por onde a mensagem passa diminui logaritmicamente, como vimos na Seção 6.3.2, pois os recursos da tabela de *fingers* estão atualizados.

8.3.3 Número de Recursos Visitados em uma Busca

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 do protocolo.

Cada ponto da Figura 8.5 representa a quantidade de recursos visitados até encontrar o recurso procurado. Por exemplo, para 1000 recursos adicionados na LBI com valores incrementais, para encontrar o recurso com valor 300 é necessário visitar 10 recursos com seis *fingers* e 14 recursos com cinco *fingers*. Esses valores são

[1]Como a LBI não tem um ponteiro ao final da lista, para adicionar um recurso no final é necessário percorrer toda a lista.

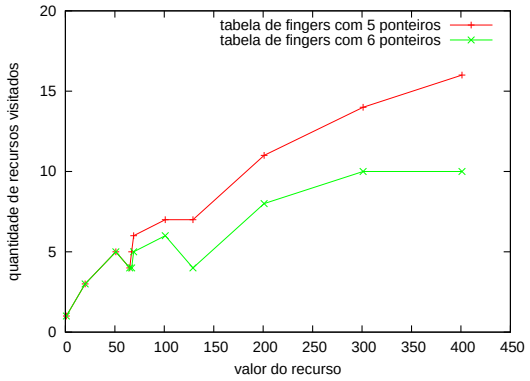


Figura 8.5: Quantidade de recursos visitados em uma busca.

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 $O(\log n)$.

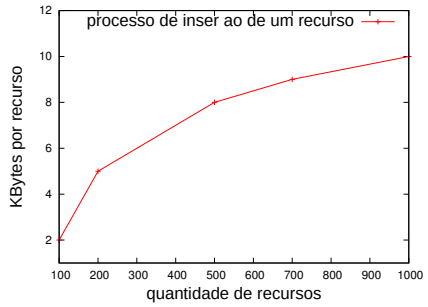
8.3.4 Largura de Banda usada pelo Protocolo

Neste experimento, adicionamos à rede 1000 pares os quais disponibilizavam os recursos que eram armazenados na LBI. Cada par disponibiliza somente um recurso, e foram adicionados um a um até completar os 1000. As medidas foram feitas quando a rede tinha 100, 500, 700 e 1000 recursos, tomando a quantidade de bytes recebidos e enviados pelos pares e dividindo pela quantidade de pares pertencentes à rede.

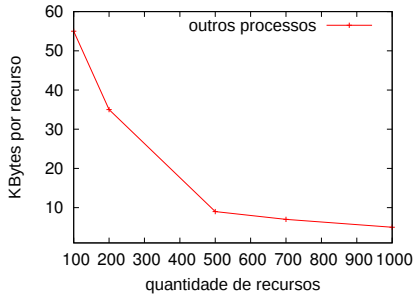
Podemos ver na Figura 8.6(a) que o consumo total de largura de banda por par, quando executada a inserção de um recurso, é uma função linear crescente, mas que não é um valor muito alto. Por exemplo, se temos 1000 recursos, o consumo médio por par será de 10 KBytes. A Figura 8.6(b) mostra o consumo de largura de banda, por par, quando executados os processos de estabilização da Seção 6.3.6. Podemos observar que a largura de banda por par diminui pois, depois que a LBI está estabilizada, esses processos não são executados. É importante destacar que, na quantidade de KBytes necessários para executar a inserção e estabilização da LBI estão incluídos também os KBytes necessários para adicionar um recurso na tabela de *hash* distribuída.

8.3.5 Buscas por Intervalo

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. Este experimento



(a) Consumo de largura de banda na inserção de um recurso



(b) Consumo de largura de banda quando executado os outros processos

Figura 8.6: Largura de banda usada pelos processos de registro e estabilização.

mede como a quantidade de recursos devolvidos em uma busca se comporta com o aumento do tempo da busca.

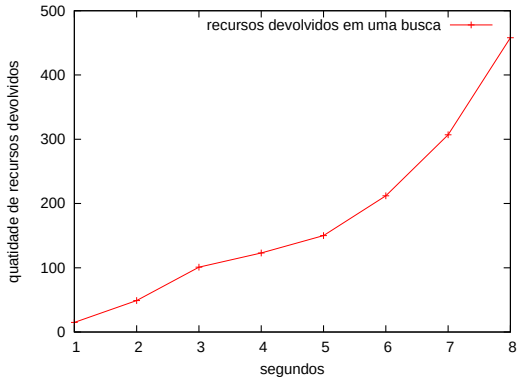


Figura 8.7: Quantidade de recursos devolvidos dada uma busca por intervalo.

Para esse experimento adicionamos 1000 recursos e fizemos uma busca por todos os valores dos recursos que pertencem à LBI. Cada ponto da Figura 8.7 representa a quantidade de recursos devolvidos em um determinado momento. Por exemplo, se a busca dura 1 segundo, serão devolvidos 15 recursos e se dura 5 segundos, serão devolvidos 150 recursos. Como o comportamento da busca (experimentalmente) corresponde a uma função exponencial (uma função linear devolveria aos 5 segundos 75 recursos) o tempo total para obter todos os valores do inter-

valo será de $O(\log n)$, obtido da fórmula: se $quantidade_{devolvidos} = O(e^{tempo})$ então $tempo_{total} = O(\log(todos_{devolvidos}))$. Com esse resultado, mantemos o desempenho das estruturas baseadas nas árvores distribuídas mostradas na próxima seção.

Capítulo 9

Integração dos Protocolos ao InteGrade

Neste capítulo será apresentado como o Protocolo de Interligação e o Protocolo de Localização podem ser integrados para tratar das requisições feitas pelos usuários do sistema InteGrade.

A seguir vamos analisar o seguinte exemplo: um usuário quer executar um programa e para isso precisa localizar 128 máquinas onde cada uma delas tenha disponível no mínimo 64 MBytes de memória RAM, 15% de utilização do processador (CPU) e 50 MBytes de disco rígido.

9.1 Visão Geral

Antes de responder como localizar os LRMs do InteGrade para executar a aplicação, vamos descrever como está composta a rede *Par-a-Par* em um momento dado. Para isso, vamos analisar a estrutura da rede segundo o Protocolo de Interligação e do Protocolo de Localização.

9.1.1 Protocolo de Interligação

Cada circulo da Figura 9.1 corresponde a um GRM e cada quadrado corresponde a um LRM. Nossa rede está composta por 100 GRMs e cada um deles possui 16 LRMs. Graças ao Protocolo de Interligação, cada GRM g possui uma vizinhança (armazenada no repositório local de pares) ordenada pela latência e que permite conhecer GRMs que estejam próximos a g.

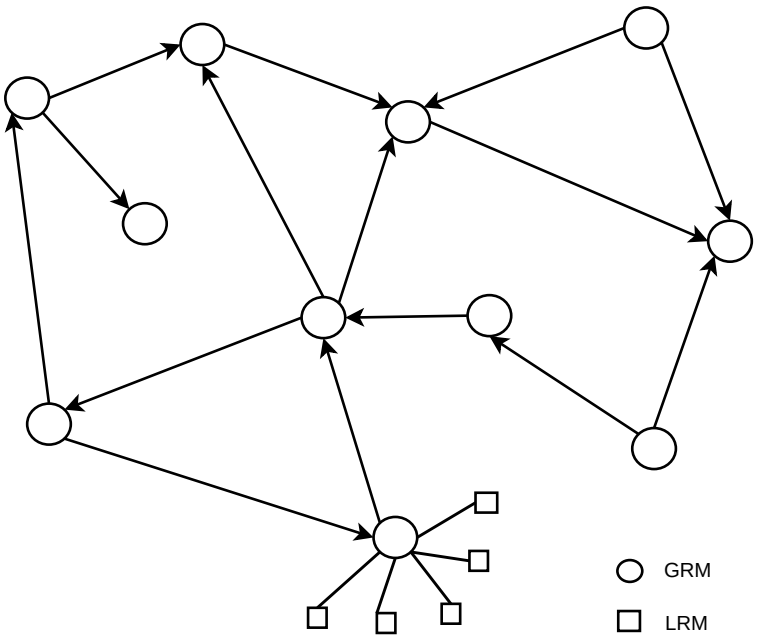


Figura 9.1: Configuração da rede formada pelos GRMs.

9.1.2 Protocolo de Localização

Para o caso do Protocolo de Localização, cada LRM que deseje compartilhar um recurso, em nosso caso a memória RAM, o processador e o disco rigido disponível, colocará o recurso na estrutura LBI e armazenada na DHT. A estrutura do recurso é definida como mostrado na Figura 9.2. A definição dos atributos da estrutura é:

- `indexKey` contém a chave com a qual o tipo de recurso foi armazenado, no nosso caso seria "RAM", "CPU" ou "HD".
- `value` corresponde ao valor numérico que o recurso disponibilizado pelo LRM apresenta em um dado momento.
- `ID` contém o identificador do LRM. O identificador pode ser o endereço IP com a porta, um certificado digital ou a IOR do LRM. O importante é que esse identificador seja único entre todos os LRMs da rede.
- `LRM` corresponde à IOR do LRM.
- `GRM` corresponde à IOR do GRM que gerencia o LRM.

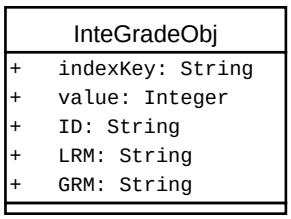


Figura 9.2: Estrutura do Recurso armazenado na LBI.

Em nosso exemplo, como são três os tipos de recursos que estão sendo compartilhados, teremos três LBIs, uma para a RAM, outra para a CPU e outra para o HD, como mostrado na Figura 9.3. É importante destacar que um LRM não necessariamente têm que estar nas três LBIs. Por exemplo, se um LRM deseja compartilhar somente a memória RAM e o CPU, mas não o HD, ele estará presente nas LBIs com chave "RAM", "CPU" e não na LBI com chave "HD".

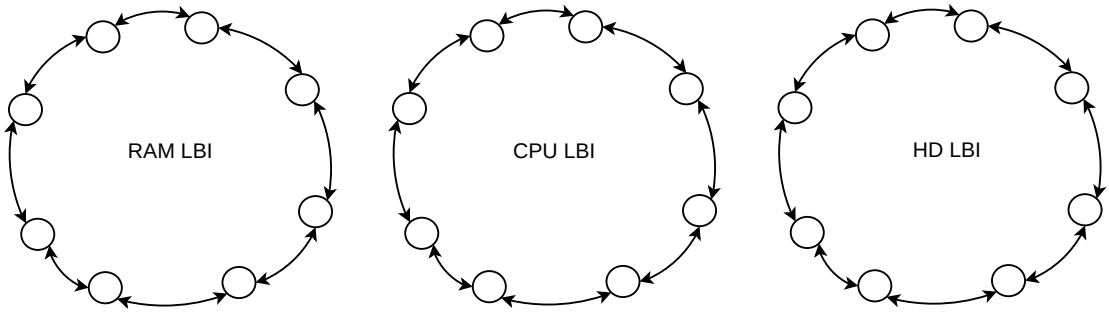


Figura 9.3: Três LBI, uma para a memória RAM, uma para o processador e outra para o disco rígido.

9.2 Obtendo os LRMs

Como mencionado no começo do apêndice, o usuário precisa encontrar 128 LRMs que cumpram com os requisitos de RAM, CPU e HD disponível. Para isso, o usuário da grade faz a requisição que é encaminhada para o GRM que está administrando o seu aglomerado. O GRM utilizará então a vizinhança e as LBI para obter os 128 LRMs. Primeiro tentando obter esses LRMs de sua vizinhança (Seção 9.2.1) e no caso de não conseguir os 128 LRMs, tenta obter os restantes das LBI (Seção 9.2.2) até completar com a quantidade total.

9.2.1 Utilizando a Vizinhança

O primeiro passo é obter o máximo número de LRMs da vizinhança, isso porque segundo nosso Protocolo de Interligação, esses LRMs estão mais próximos em termos de latência e portanto a comunicação entre eles será mais eficiente.

Para isso, faz-se uma requisição aos GRMs definidos nas primeiras entradas do repositório local de pares, os quais devolverão os LRMs disponíveis no momento. Isto é possível pois, segundo explicado na Seção 2.1, o GRM é responsável por coletar várias informações dos LRM e portanto conhece quais os LRMs que cumprem com os requisitos da busca.

9.2.2 Utilizando as LBIs

A Seção anterior obtém os LRMs da vizinhança, mas ainda assim pode ser que a quantidade de LRMs necessários para executar uma aplicação não seja atingida. Para completar com os LRM que faltam, utilizaremos as LBIs apresentadas na Seção 9.1.2.

Na Figura 9.4 vemos o Algoritmo utilizado para obter os LRMs, das LBIs, que atendem os requisitos de uma certa requisição. Os passos do algoritmo são:

1: **ObterLRMs()**
2: $grmHash \leftarrow \emptyset$;
3: $head\_ram \leftarrow$ dht.super_head("RAM");
4: $listaRAM \leftarrow$ dht.SEARCH(64, $INFINITO$, $head\_ram$);
5: $head\_cpu \leftarrow$ dht.super_head("CPU");
6: $listaCPU \leftarrow$ dht.SEARCH(15, 100, $head\_cpu$);
7: $head\_hd \leftarrow$ dht.super_head("HD");
8: $listaHD \leftarrow$ dht.SEARCH(50, $INFINITO$, $head\_hd$);
9: $lista\_lrm\_validos \leftarrow$ filtraLista($listRAM$, $listaCPU$, $listaHD$);
10: indexa($grmHash$, $lista\_lrm\_validos$);
11: **devolve** $grmHash$;

Figura 9.4: Algoritmo utilizado para buscar os LRMs nas LBIs dado um requisito para a execução de uma tarefa.

1. Obter da LBI, com chave RAM, os LRMs que tenham uma memória RAM disponível maior que 64 MBytes. (Linha 4).

2. Obter da LBI, com chave CPU, os LRMs que tenham uma porcentagem de uso de processador disponível maior que 15. (Linha 6).

3. Obter da LBI, com chave HD, os LRMs que tenham uma quantidade de disco rígido disponível maior que 50 MBytes. (Linha 8).

4. Fazer um emparelhamento entre os LRM das listas obtidas e devolver somente os que estão contidos nas três LBIs (Linha 9). Se o LRM não pertencer a uma LBI quer dizer que não cumpre com todos os requisitos.

5. Indexar em uma estrutura (*hash*) esses LRMs válidos pelo GRM que o gerencia. Com isso, poderemos observar quantos LRMs têm o mesmo GRM (Linha 10).

6. Devolver ao usuário a estrutura *hash* que poderá escalonar a aplicação de diferentes maneiras (Linha 11). Por exemplo, uma heurística é utilizar os LRMs que estejam gerenciados pelo GRM que tenha o maior número de LRMs.

Capítulo 10

Conclusões e Trabalhos Futuros

Após a descrição dos protocolos e dos resultados obtidos no desenvolvimento deste trabalho, apresentamos as conclusões e os trabalhos futuros que poderão ser desenvolvidos com base nesta pesquisa.

10.1 Conclusões

O objetivo deste trabalho foi desenvolver protocolos *Par-a-Par* que permitam estruturar os pares de uma rede, baseado na latência entre eles, e localizar os recursos disponibilizados por esses pares. Esses protocolos, foram desenvolvidos visando sua utilização em sistemas de Computação em Grade, como o InteGrade.

Acreditamos que tais objetivos foram atingidos: nossos protocolos podem se adequar aos sistemas de Computação em Grade por serem independentes de um administrador central, possuírem suporte para buscas dos recursos baseadas em seus atributos e manterem a escalabilidade.

A adoção de tecnologias *Par-a-Par* já implementadas como base para a comunicação e localização de um determinado recurso nos pares da rede se mostrou muito eficiente considerando o tempo e a facilidade de desenvolvimento do protocolo. A estrutura utilizada foi a tabela de *hash* distribuída e a implementação usada foi o Bamboo. Bamboo permitiu, de forma simples através de sua API simular recursos em redes de grande área sem termos que nos preocupar com essa implementação e funcionamento.

Sobre o Protocolo de Interligação, o principal objetivo é conectar os pares de forma que a comunicação entre eles, em termos de latência, seja a mais rápida possível. A utilização da camada de rede, para encontrar novos pares, foi uma ajuda

muito grande em dois sentidos: primeiro, a obtenção dos caminhos entre dois pares já foi feita e calculada por essa camada e segundo, existem muitos caminhos entre os pares que se entrecruzam.

Para explicar isso, no primeiro ponto devemos destacar que, mesmo que a camada de rede não consiga obter "o melhor caminho" entre dois pares, ela consegue obter um bom caminho com o qual começar a procura por novos pares sem ter que implementar essa funcionalidade. No segundo ponto, quando os caminhos se entrecruzam, esses cruzamentos correspondem geralmente aos roteadores intermediários obtidos da camada de rede os quais, em nosso protocolo, foram utilizados para armazenar o par mais próximo desse roteador.

Com a simulação desse protocolo, podemos concluir que o tempo gasto para ingresso e atualização da rede é constante. Outro ponto importante é que na primeira iteração para atualizar a rede já se tinha uma estrutura onde a conexão entre os pares era muito boa. Finalmente, a quantidade de largura de banda consumida é linear em relação ao número de pares, devido ao uso da tabela de *hash* distribuída. Mesmo com essa deficiência, investigações sobre o desempenho dessa estrutura asseguram que ela é totalmente escalável para centena de milhares de pares conectados.

Sobre o Protocolo de Localização, o objetivo é permitir a busca de recursos por um intervalo de valores, de forma eficiente e utilizando a tabela de *hash* distribuída. Para isso, foi proposta uma estrutura simples e eficiente baseada em uma lista distribuída e ordenada, para o registro e buscas dos recursos. Esses recursos podem possuir valores repetidos e, pelo nosso conhecimento, este é o primeiro trabalho que resolve esse problema.

De acordo com os resultados obtidos nas simulações deste protocolo, a nossa estrutura é tão eficiente quanto as propostas que usam árvores distribuídas, com a vantagem de utilizar uma estrutura mais simples de administrar.

10.2 Trabalhos Futuros

Pesquisas a serem realizadas a partir deste trabalho podem se dividir em duas áreas: na interligação dos pares e na localização dos recursos disponibilizados por eles.

Sugestões de melhoria no Protocolo de Interligação dizem respeito à largura de banda consumida e a integração da tabela de *hash* distribuída com um sistema

de Computação em Grade.

A largura de banda consumida pode ser diminuída se não utilizarmos a tabela de *hash* distribuída. Com essa abordagem, o acesso aos objetos roteadores que possibilitam conhecer novos pares, será uma tarefa difícil de conseguir.

Em relação à integração do nosso protocolo (baseado em uma implementação de uma tabela de *hash* distribuída) com um sistema de Computação em Grade, ela pode ser feita somente no nível de comunicação. Para isso podemos utilizar sistemas de middleware, como CORBA, que permitem a comunicação entre objetos através de chamadas remotas.

Considerando o Protocolo de Localização, existem algumas pesquisas que podem ser efetuadas. Por exemplo, o custo da inserção pode ser diminuído usando alguns recursos que sejam réplicas dos identificadores mais visitados.

Outro análise pode ser na quantidade de troca de mensagens para realizar uma busca por intervalo. Ela pode ser diminuída utilizando a propagação dos pares que já foram visitados em uma requisição. Isto pode trazer problemas de largura de banda consumida que terão que ser analisados.

Finalmente, podemos sugerir uma pesquisa no compartilhamento de arquivos de dados em Grades Computacionais utilizando técnicas *Par-a-Par*. Para isso, seria interessante que a transferência desses arquivos fosse com os aglomerados mais próximos em termos de latência.

Referências Bibliográficas

[AS03] James Aspnes and Gauri Shah. Skip Graphs. In *SODA '03: Proceedings of the Fourteenth Annual ACM SIAM Symposium on Discrete Algorithms*, 2003.

[ASSW03] Tom Anderson, Scott Shenker, Ion Stoica, and David Wetherall. Design guidelines for robust Internet protocols. *SIGCOMM Computer Communication Review*, 33(1):125–130, 2003.

[AX02] Artur Andrzejak and Zhichen Xu. Scalable, Efficient Range Queries for Grid Information Services. In *Proceedings of the Second IEEE International Conference on Peer-to-Peer Computing*, 2002.

[Bam] The Bamboo Distributed Hash Table. `http://www.bamboo-dht.org`. Último acesso em Junho/2005.

[BBK02] Magdalena Balazinska, Hari Balakrishnan, and David Karger. INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery. In *International Conference on Pervasive Computing 2002*, Zurich, Switzerland, August 2002.

[BV04] Rajkumar Buyya and Srikumar Venugopal. The Gridbus Toolkit for Service Oriented Grid and Utility Computing: An Overview and Status Report. In *Proceedings of the First IEEE International Workshop on Grid Economics and Business Models (GECON)*, pages 19–36. IEEE Press, Abril 2004.

[BZH03] Ali Raza Butt, Rongmei Zhang, and Y. Charlie Hu. A self-organizing flock of condors. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. IEEE Computer Society, 2003.

[CFFK01] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid Information Services for Distributed Resource Sharing. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed*

Computing, page 181, Washington, DC, USA, 2001. IEEE Computer Society.

[Cho] The Chord Project. `http://www.pdos.lcs.mit.edu/chord`. Último acesso em Junho/2005.

[CJK+01] Miguel Castro, Michael B. Jones, Anne-Marie Kermarrec, Antony Rowstron, Marvin Theimer, Helen Wang, and Alec Wolman. An Evaluation of Scalable Application-level Multicast Built Using Peer-to-peer overlays. In *ACM SIGCOMM 2001*, Agosto 2001.

[CLL02] Jacky Chu, Kevin Labonte, and Brian Neil Levine. Availability and Locality Measurements of Peer-to-Peer File Systems. In *Proceedings of ITCom: Scalability and Traffic Control in IP Networks II Conferences*, 2002.

[Con] The Condor Project. `http://www.cs.wisc.edu/condor`. Último acesso em Julho/2005.

[Cos97] P. R. Cosway. Replication control in distributed b-trees. Technical report, Cambridge, MA, USA, 1997.

[dlB59] Renee de la Briandais. File Searching Using Variable Length Keys. In *Proceedings of the Western Joint Computer Conference*, volume 15, 1959.

[eA01] Nelson Minar et Al. *Peer-to-Peer – Harnessing the Power of Disruptive Technologies*. O'Reilly, 2001.

[FK97] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[GDS+03] Krishna P. Gummadi, Richard J. Dunn, Stefan Saroiu, Steven D. Gribble, Henry M. Levy, and John Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 314–329, New York, NY, USA, 2003. ACM Press.

[GKG+04] Andrei Goldchleger, Fabio Kon, Alfredo Goldman, Marcelo Finger, and Germano Capistrano Bezerra. InteGrade: Object-Oriented Grid Middleware Leveraging Idle Computing Power of Desktop Machines. *Concurrency and Computation: Practice and Experience.*, 16:449–459, 2004.

[Glo] The Globus Alliance. http://www.globus.org. Último acesso em Abril/2005.

[gnua] Gnutella 2 Protocol. http://www.gnutella2.com/index.php/Main_Page#The_Protocol. Último acesso em Julho/2005.

[gnub] Gnutella Project. http://www.gnutella.com. Último acesso em Junho/2005.

[Gol04] Andrei Goldchleger. InteGrade: a Middleware System for Opportunistic Grid Computing. Master's thesis, Department of Computer Science – University of São Paulo, Dezembro 2004. in Portuguese.

[gri] The Gridbus Project. http://www.gridbus.org. Último acesso em Junho/2005.

[Gro02] Object Management Group. CORBA v3.0 Specification. OMG Document 02-06-33, 2002.

[GS04] Jun Gao and Peter Steenkiste. An Adaptive Protocol for Efficient Support of Range Queries in DHT-Based Systems. In *Proceedings of the 12th IEEE International Conference on Network Protocols (ICNP2004)*, 2004.

[HHB+03] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham Boon, Thau Loo, Scott Shenker, and Ion Stoica. Querying the Internet with PIER. In *Proceedings of 19th International Conference on Very Large Databases.*, sep 2003.

[IFN02] Adriana Iamnitchi, Ian Foster, and Daniel Nurmi. A peer-to-peer approach to resource location in grid environments. In *Proceedings of the 11 th IEEE International Symposium on High Performance Distributed Computing*, page 419, Washington, DC, USA, 2002. IEEE Computer Society.

[Int] InteGrade. http://integrade.incubadora.fapesp.br. Último acesso em Dezembro/2005.

[Jun04] Jun Gao and Peter Steenkiste. An Adaptive Protocol for Efficient Support of Range Queries in DHT-based Systems. *12th IEEE International Conference on Network Protocols (ICNP 2004)*, 2004.

[jxt] JXTA. http://www.jxta.org. Último acesso em Março/2005.

[Kat02] Daishi Kato. GISP: Global Information Sharing Protocol. A Distributed Index for Peer-to-Peer Systems. In *P2P '02: Proceedings of the Second International Conference on Peer-to-Peer Computing*, page 65. IEEE Computer Society, 2002.

[kaz] Kazaa. `http://www.kazaa.com`. Último acesso em Julho/2005.

[KCM+00] Fabio Kon, Roy H. Campbell, M. Dennis Mickunas, Klara Nahrstedt, and Francisco J. Ballesteros. 2K: A Distributed Operating System for Dynamic Heterogeneous Environments. In *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing (HPDC'9)*, pages 201–208, Pittsburgh, Agosto 2000.

[Knu73] D. E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching.* Addison-Wesley, 1973. Section 6.2.4.

[KR04] David R. Karger and Matthias Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *SPAA '04: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 36–43. ACM Press, 2004.

[Leu02] Bo Leuf. *Peer to Peer.* Addison-Wesley, 2002.

[LLM88] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, Junho 1988.

[Lui04] Luis Garces-Erice and Pascal A. Felber and Ernst W. Biersack and Guillaume Urvoy-Keller and Keith W. Ross. Data Indexing in Peer-to-Peer DHT Networks. *In Proceedings of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 200–208, 2004.

[Min01] Nelson Minar. Distributed Systems Topologies, 2001. `http://www.openp2p.com/pub/a/p2p/2001/12/14/topologies_one.html`.

[nap] Napster. `http://www.napster.com`. Último acesso em Julho/2005.

[Pos81] J. Postel. Internet Protocol - DARPA Internet Program Protocol Specification RFC 792, 1981. `http://www.ietf.org/rfc/rfc792.txt`.

[Pug89] William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. In *Workshop on Algorithms and Data Structures*, pages 437–449, 1989.

[RD01] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350, London, UK, 2001. Springer-Verlag.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 161–172, 2001.

[RGRK04] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling Churn in a DHT. In *USENIX Annual Technical Conference*, Junho 2004.

[RHS03] S. Ratnasamy, J. Hellerstein, and S. Shenker. Range Queries over DHTs. In *Technical Report IRB-TR-03-009, Intel Corp.*, june 2003.

[Rit01] Jordan Ritter. Why Gnutella Can't Scale, 2001. `http://www.darkridge.com/~jpr5/doc/gnutella.html`.

[RLS+03] Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, and Ion Stoica. Load Balancing in Structured P2P Systems. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS '03).*, 2003.

[RR05] Vladimir Rocha and Eric Ross. Uma Estrutura Escalável e Eficiente Para buscas por Intervalo sobre DHTs nas Redes P2P. In *WP2P - I Peer-to-Peer Workshop of the Brazilian Symposium on Computer Networks (SBRC)*, Maio 2005.

[SGG02] S. Saroiu, P. Gummadi, and S. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *Proceedings of Multimedia Computing and Networking*, 2002.

[SMK+03] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Infocom'03*, Abril 2003.

[Sta03] William Stallings. *Data and Computer Communication*. Maxwell Macmillan International, seventh edition, 2003.

[STDG01] Clay Shirky, Kelly Truelove, Rael Dornfest, and Lucas Gonze. *P2P Networking : The Emergent P2P Platform of Presence, Identity, and Edge Resources*. O'Reilly, 2001.

[Tan02] Andrew Tanenbaum. *Computer Networks*. Prentice Hall, 2002.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[VYW+02] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. Scalability and Accuracy in a LargeScale Network Emulator. In *Proceedings 5th OSDI, Dec. 2002.*, 2002.

[YB04] Praveen Yalagandula and James C Browne. Solving Range Queries in a Distributed System. In *Technical Report TR-04-18, Department of Computer Science, The University of Texas at Austin*, may 2004.

[YVB03] Jia Yu, Srikumar Venugopal, and Rajkumar Buyya. A Market-Oriented Grid Directory Service for Publication and Discovery of Grid Service Providers and their Services. Technical report, Grid Computing and Distributed Systems (GRIDS) Laboratory, The University of Melbourne, Australia, Janeiro 2003. GRIDS-TR-2003-0.

[ZS04] X. Zhang and J. Schopf. Performance Analysis of the Globus Toolkit Monitoring and Discovery Service, MDS2. In *Proceedings of the International Workshop on Middleware Performance (MP 2004), part of the 23rd International Performance Computing and Communications Conference*, 2004.