UNIVERSIDADE FEDERAL FLUMINENSE
INSTITUTO DE COMPUTAÇÃO
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Pedro Henrique Rigony Monteiro

Mecanismos de Busca em Redes Cooperativas Não Estruturadas

Niterói
2006

PEDRO HENRIQUE RIGONY MONTEIRO

Mecanismos de Busca em Redes Cooperativas Não Estruturadas

Monografia apresentada ao Departamento de Ciência da Computação da Universidade Federal Fluminense como parte dos requisitos para obtenção do Grau de Bacharel em Ciência da Computação

Orientador: Célio Vinicius Neves de Albuquerque

Niterói

2006

PEDRO HENRIQUE RIGONY MONTEIRO

Mecanismos de Busca em Redes Cooperativas Não Estruturadas

Monografia apresentada ao Departamento de Ciência da Computação da Universidade Federal Fluminense como parte dos requisitos para obtenção do Grau de Bacharel em Ciência da Computação

Aprovado em dezembro de 2006.

BANCA EXAMINADORA

_____

Prof. CÉLIO VINICIUS NEVES DE ALBUQUERQUE
Orientador
UFF

_____

Profa. ANNA DOLEJSI SANTOS
UFF

_____

Profa. MARIA CRISTINA SILVA BOERES
UFF

Niterói
2006

RESUMO

Com o surgimento das aplicações de compartilhamento de arquivos na Internet surge o desafio de como implementar um mecanismo de busca eficiente que minimize o atraso fim-a-fim de comunicação entre os participantes de uma rede cooperativa não estruturada e, ao mesmo tempo, seja escalável, visto que esse tipo de rede pode suportar milhares de usuários simultaneamente enviando e recebendo arquivos.

Este trabalho descreve, implementa em Java e analisa dois mecanismos diferentes, no nível de aplicação, para a busca de palavras chave entre usuários de uma rede cooperativa não estrutrada, com o objetivo de comparar suas características, demonstrando como se podem minimizar os problemas enfrentados utilizando-se protocolos e arquiteturas de comunicação existentes.

Palavras Chave:

Internet, Redes Cooperativas Não Estruturadas, Mecanismos de Busca.

ABSTRACT

With the advent of file sharing applications on the Internet comes the challenge to provide support to new search mecanisms, at the application level, that minimizes the end-to-end delay and provides scalability.

This work describes, implements and analyses two different mecanisms at the application level, for keyword searching over unstructured peer-to-peer networks, with the objective of comparing their features, showing how occurring problems can be minimized by using existing communication protocols and architectures.

Keywords:

Internet, Unstructured Peer-to-Peer Networks, File Search Mecanisms.

LISTA DE ACRÔNIMOS

API:	*Application Programming Interface* (ou Interface de Programação de Aplicativos).
ASCII:	*American Standard Code for Information Interchange*
IP:	*Internet Protocol*
P2P:	*Peer-to-Peer*
TCP:	*Transmission Control Protocol*
TTL:	*Time-to-Live*

SUMÁRIO

LISTA DE FIGURAS

CAPÍTULO 1 - INTRODUÇÃO

1.1 Motivação

Desde o surgimento das redes computacionais, acontecem trocas de dados entre máquinas fisicamente distantes entre si. Neste primeiro momento, o único paradigma existente para que isto fosse possível era o modelo cliente/servidor.

Neste paradigma, ilustrado na Figura 1, o servidor é uma máquina dedicada e geralmente com grande disponibilidade de espaço para armazenamento. A busca de conteúdo e toda a computação necessária para o processamento das requisições estão no servidor.

Já os clientes são os usuários que requisitam conteúdo dos servidores e estão, normalmente, em maior número.

Neste paradigma existe um forte problema com a centralização do fluxo de envio de dados, pois caso muitos clientes requisitem arquivos ao mesmo tempo, o servidor poderá ficar sobrecarregado.

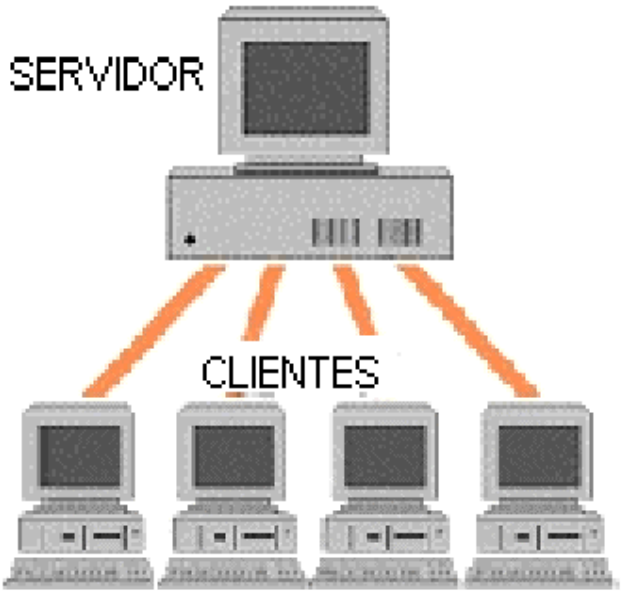

Figura 1: Modelo Cliente/Servidor

Para minimizar o problema, surgiu um novo paradigma: a troca de dados direta entre clientes, ou uma rede cooperativa.

Arquiteturas de rede chamadas cooperativas ou *Peer-to-Peer* (P2P) são caracterizadas pela troca direta de recursos computacionais em vez de através do intermédio de um servidor centralizado. Pode-se observar a ligação entre os participantes da rede na Figura 2.

Os participantes de uma rede cooperativa se conectam inicialmente à rede através de um servidor central ou através de nós inicializadores. Esses conceitos serão revistos adiante.

A motivação de se fazer aplicações P2P vem da sua grande habilidade em funcionar, escalar e se organizar na presença de uma população com taxas de conexão e desconexão muito elevadas, falhas da rede e dos recursos computacionais sem a necessidade de um servidor centralizado e da sobrecarga de administração do sistema.

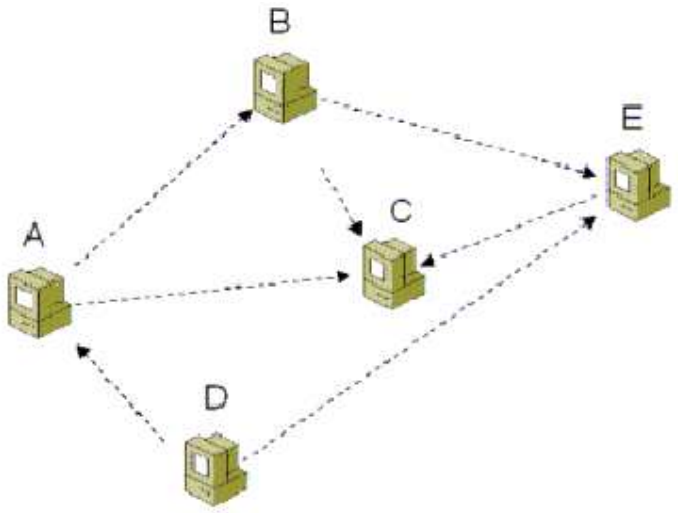

Figura 2: Modelo P2P

Tais arquiteturas tipicamente possuem como características a escalabilidade, resistência à censura e controle central, aumento de acesso a recursos. A administração, manutenção, responsabilidade pela operação e mesmo o conceito de posse em sistemas P2P são distribuídos entre os usuários em vez de serem controlados por uma única empresa ou pessoa.

A escalabilidade é caracterizada pela manutenção da performance do sistema independentemente do número de nós ou documentos na rede. Um aumento drástico no número de nós ou documentos terá um pequeno impacto na performance e disponibilidade de recursos.

Existem dois tipos principais de arquiteturas para redes cooperativas: estruturada e não estruturada.

Em uma rede estruturada, a posição dos arquivos está fortemente relacionada com a organização da camada de rede P2P. Quando um arquivo é inserido na rede, este é colocado numa posição precisamente especificada de modo a formar um mapeamento entre conteúdo e localização do mesmo.

Já em uma rede não estruturada, a posição dos arquivos não está relacionada à organização da camada de rede P2P. Numa rede não estruturada, o conteúdo precisa ser localizado. Mecanismos de busca variam desde métodos de força bruta, como inundar a rede com buscas até localizar o conteúdo a estratégias mais sofisticadas que incluem a busca por passos aleatórios e o uso de índices de roteamento.

Sistemas não estruturados são geralmente mais apropriados para acomodar grande variação na população de usuários.

Este trabalho utilizará os conceitos das redes não estruturadas, como a rede Gnutella [6] tendo em vista seu grande número de usuários e a necessidade de se aperfeiçoar a localização de conteúdos neste tipo de rede.

O mecanismo inicial implementado faz uma busca radial, ou seja, cada nó da rede propaga a busca para todos os seus vizinhos na rede, decrementando um campo de tempo de vida a cada nova busca propagada. Assim, criam-se subredes dentro de uma rede maior, visto que cada busca tem um alcance limitado.

Outro problema dessa abordagem é o de escalabilidade, pois cada busca gera mensagens distintas para todos os vizinhos, que por sua vez, ao receberem a mensagem também geram novas mensagens para todos os vizinhos. Logo, caso o número de usuários cresça muito, teremos muitas mensagens trafegando na rede ao mesmo tempo e diminuindo exponencialmente sua performance proporcionalmente ao número de novas conexões [3].

A motivação para este trabalho é propor um mecanismo de busca para redes cooperativas não estruturadas que diminua o tempo de espera em uma busca, seja escalável e atinja melhores resultados do que as buscas já existentes.

Na implementação do mecanismo, decidiu-se construir uma aplicação de busca de palavras-chaves em uma rede cooperativa não estrutrada e descentralizada, desenvolvida em linguagem Java, criada pela *Sun Microsystem* [2].

A construção desta aplicação permite uma comparação entre o método radial e um método otimizado, que diminua os problemas relatados acima e permita que outras aplicações possam ser construídas tirando proveito dos resultados obtidos.

1.2 Objetivos

O objetivo deste projeto é propor um mecanismo de busca que aumente a escalabilidade e seja mais eficiente que o mecanismo de busca radial, ambos implementados em uma aplicação Java para comparação posterior de performance.

Esse mecanismo difere-se da busca radial por consultar usuários cadastrados em uma lista de afinidades, que provavelmente têm o mesmo gosto por conteúdos e assim melhorar a eficiência da busca. A cada arquivo enviado, o remetente guardará o requisitante numa lista de afinidades, para que posteriormente possa consultá-la.

A escalabilidade também será aumentada, pois existirão apenas dois vizinhos por nó conectado, limitando o impacto do espalhamento de mensagens trafegando por muitos nós simultâneamente e, assim, utilizando menos banda passante e recursos computacionais dos participantes da rede.

A expectativa para este trabalho é que novas aplicações possam utilizá-lo, facilitando o compartilhamento de arquivos e dando um incentivo ao crescimento das redes cooperativas.

E ainda, o projeto servirá como base para estudos posteriores que possam tirar proveito das idéias apresentadas e chegar a um algoritmo com melhores resultados.

1.3 Organização

Este trabalho foi organizado da seguinte forma. No Capítulo 2 são explicados os princípios de redes cooperativas e as variações existentes. O Capítulo 3 descreve os mecanismos de busca existentes e os implementados. No Capítulo 4 são detalhados os algoritmos utilizados na implementação. No Capítulo 5, será feita uma análise de desempenho do mecanismo de busca por afinidades. E finalmente, o Capítulo 6, onde se tiram conclusões sobre a possível utilização do trabalho produzido.

CAPÍTULO 2 – PRINCÍPIOS DE REDES COOPERATIVAS

2.1 Tipos de Redes Cooperativas

Como mostrado no capítulo anterior, as redes cooperativas são classificadas em dois tipos principais: redes estruturadas e não-estruturadas.

2.2 Redes Estruturadas

As redes estruturadas apresentam forte ligação entre conteúdo e sua localização. Um exemplo de rede estruturada é a Freenet [4].

Nesta rede, os arquivos são identificados por uma chave binária única. Três tipos de chaves são aceitas, a mais simples é baseada numa função *hash* aplicada sobre uma *string* de descrição do arquivo.

Quando um nó se junta à Freenet, ele envia mensagens de inserção de dados para espalhar seus arquivos pela rede e aumentar a disponibilidade de conteúdo (na verdade ele envia ponteiros para os seus arquivos), conforme ilustrado na Figura 3.

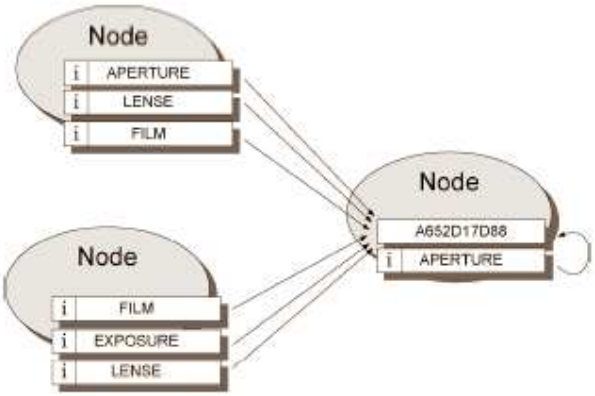

Figura 3: Arquitetura Freenet

Os nós da Freenet se identificam enviando uma mensagem de estabelecimento de conexão uns para os outros. Se um nó recebe esta mensagem e está ativo, ele envia uma mensagem de resposta a esse estabelecimento de conexão especificando a versão do protocolo com a qual ele pode se comunicar. Essas mensagens são guardadas por algumas horas. Outras transações subseqüentes entre os mesmos usuários durante este tempo podem omitir esses passos de comunicação.

Para encontrar um arquivo dentro da Freenet, o usuário requisitante envia uma mensagem de busca especificando o identificador da transação, o tempo de vida, a profundidade e a palavra chave da busca.

O nó que recebe o pedido irá procurar pela chave requisitada em seus arquivos e se não encontrar, repassará a busca para outros vizinhos na rede. Após o tempo determinado inicialmente pelo requisitante se esgotar, ele vai assumir que houve um fracasso.

Se a requisição for bem sucedida, o nó que encontrou o arquivo vai responder com uma mensagem de arquivo encontrado contendo o endereço da máquina onde o conteúdo foi localizado e o tamanho do arquivo.

Logo depois, o próprio arquivo é enviado em outra mensagem.

As redes estruturadas oferecem uma solução bastante escalável para buscas por palavras exatas, ou buscas pelo identificador único pelo qual o arquivo é conhecido.

Uma grande desvantagem deste tipo de rede é a grande dificuldade de se manter a estrutura necessária para roteamento das buscas quando existe uma população de usuários com altas taxas de conexão e desconexão.

2.3 Redes Não-Estruturadas

As redes não-estruturadas são utilizadas por milhões de usuários espalhados pelo mundo inteiro e motivaram esse trabalho, pois a melhoria dos métodos de busca pode proporcionar um aumento ainda maior no desempenho das trocas de conteúdo e facilitar a vida de todos.

As redes não-estruturadas também podem ser subdivididas em algumas classificações: centralizada, parcialmente centralizada e totalmente descentralizada.

2.3.1 Redes Centralizadas

Em uma rede centralizada, todos os clientes se conectam a um servidor centralizado que mantem uma tabela de conexões de usuários e uma tabela listando os arquivos que cada usuário possui e deseja compartilhar na rede, junto com descrições dos arquivos (metadados).

As vantagens desse tipo de sistema são que ele é fácil de implementar e localiza arquivos com rapidez e eficiência.

Suas desvantagens são que esse tipo de sistema é vulnerável a censura, processos legais e falhas técnicas, já que para acessarmos o conteúdo temos que passar pelo controle do servidor. Além disso, esses sistemas são considerados inerentemente não escaláveis, pois

estão sujeitos a limitações de tamanho do banco de dados e capacidade de responder a buscas. Um exemplo de arquitetura centralizada é o Napster [4], um dos primeiros aplicativos de compartilhamento de arquivos, cuja interface é ilustrada na Figura 4.

O protocolo do Napster é dividido em comunicações “cliente/servidor” e “cliente/cliente”. Cada mensagem é composta por três campos: *lenght*, *type* e *data*. O campo *lenght* especifica o tamanho em bytes da porção de dados da mensagem. O campo *type* especifica o tipo da mensagem. Já o campo *data* é composto por uma *string* em ASCII.

Em vez de descobrir os vizinhos utilizando mensagens de ping ou aperto de mãos utilizados nas redes Gnutella e Freenet, um usuário do Napster se loga a um servidor que é responsável por manter uma lista de usuários conectados e um diretório de conteúdos.

Para se conectar ao servidor, cada máquina envia uma mensagem de *login* para o servidor informando seu apelido, senha e número da porta aberta para troca de arquivos. Então o servidor vai responder com uma mensagem de aceite de login para indicar o sucesso na comunicação.

Para buscar e baixar o arquivo, cada usuário deve enviar uma mensagem para o servidor, que responde uma mensagem contendo algumas informações, tais como: o apelido do usuário que possui o arquivo, o nome do arquivo e o tamanho do arquivo. Para requistar o *download*, uma mensagem de requisição de *download* é enviada para o servidor.

O servidor vai responder com uma mensagem de aceite do *download*, que contém informações mais detalhadas. Se a mensagem indica que a porta da máquina que contém o arquivo é igual a zero, o requisitante deve enviar uma mensagem de requisição alternativa para o servidor para pedir que o arquivo seja enviado. Neste caso, ele espera que o remetente se conecte à sua porta aberta para compartilhamento Napster.

Para aceitar a conexão, o usuário que contém o arquivo envia um caracter “1” em ASCII. Quando o requisitante recebe o caracter, ele vai enviar uma mensagem de requisição para o arquivo que deseja baixar contendo a *string* “*GET*” em um único pacote e depois enviando mais um pacote contendo o apelido, o nome do arquivo e o deslocamento em relação ao início do arquivo de onde ele deseja começar a receber.

Então o remetente envia uma mensagem contendo o tamanho do arquivo ou uma mensagem de erro. Imediatamente após o tamanho começa o fluxo de dados do arquivo

Figura 4: Interface do Napster

2.3.2 Redes Descentralizadas

Em uma rede descentralizada, não existe coordenação central das atividades na rede e usuários se conectam uns aos outros diretamente através de uma aplicação que funciona como cliente e servidor.

Um exemplo desta arquitetura é a rede Gnutella, que utiliza IP como serviço de rede e possui quatro tipos de mensagens: PING (uma requisição para saber se um nó está ativo), PONG (resposta ao PING contendo endereço IP, porta do computador, número e tamanho dos arquivos que estão sendo compartilhados), Query (mensagem de busca contendo uma string com a palavra-chave e o requerimento mínimo de velocidade para o nó que vai responder à mensagem) e Query Hits (resposta à Query contendo o número IP, porta e velocidade do computador, número de arquivos que correspondem à busca e o ponteiro para o arquivo).

A arquitetura original do Gnutella usa um mecanismo de espalhamento de *pings* e *queries*. Cada nó que recebe uma mensagem a repassa a todos os seus vizinhos na rede. As mensagens de resposta são roteadas pelo caminho inverso ao da requisição anterior. Para limitar o volume de tráfego, cada mensagem contém um campo time-to-live (TTL). A cada "pulo" o valor desse campo é decrementado e quando chega a zero a mensagem é descartada.

A fim de eliminar loops com mensagens duplicadas, cada mensagem contém um identificador único.

Um dos problemas do campo de time-to-live é que essa prática segmenta a rede em sub-redes impondo a cada nó um horizonte virtual além do qual nenhuma de suas mensagens pode alcançar.

2.3.3 Redes Parcialmente Centralizadas

Essa arquitetura utiliza o conceito de supernós: nós que são dinamicamente escolhidos para realizarem a tarefa de um servidor centralizado para um grupo pequeno de nós. Os supernós são eleitos se possuem largura de banda e poder de processamento suficientes.

Existem duas grandes vantagens nesses sistemas: o tempo de busca é reduzido em comparação com sistemas descentralizados e não possuem um ponto único suscetível à falhas.

Outra vantagem é que a heterogeneidade das redes P2P é explorada de modo útil: os supernós (com maior capacidade) pegam grande parte da carga da rede, enquanto que os nós comuns não ficam sobrecarregados.

Um exemplo de aplicação que utiliza este tipo de rede é o Kazaa [7] e suas variações, cuja interface é ilustrada na Figura 5.

Figura 5: Interface do Kazaa

2.4 Evolução das Arquiteturas Não-Estruturadas

Novos estudos e implementações estão sendo feitos para melhorar as arquiteturas de redes P2P não-estruturadas.

Um deles é substituir a inundação de *queries* por caminhadas aleatórias paralelas nas quais cada nó escolhe um nó aleatório e propaga a busca apenas por ele. Essa medida tem o poder de aumentar a escalabilidade, pois menos mensagens vão trafegar pela rede simultaneamente [5].

Outro recurso que pode ser empregado é o uso de índices locais: estruturas de dados nas quais cada nó mantem índices dos dados de outros nós distantes de até um determinado raio a partir dele. Isso pode diminuir o tempo necessário para buscar um conteúdo na rede [3].

Outra tendência é usar o passado de cada nó para propagar as *queries* utilizando um *ranking* de nós para os quais o maior número de buscas foram bem sucedidas [3].

O estudo feito para este trabalho é baseado no último apresentado (escolha prioritária dos nós para os quais a busca foi bem sucedida) e cada busca bem sucedida resultará em uma lista de usuários preferenciais, que serão consultados antes dos usuários comuns.

CAPÍTULO 3 – ALGORITMOS DE BUSCA

3.1 Algoritmos de Busca em Redes Não-Estruturadas

Para introduzir os algoritmos implementados para este trabalho, serão apresentados algoritmos da rede Gnutella, cuja arquitetura foi a que inspirou o projeto.

A busca comumente utilizada pela rede Gnutella é baseada numa propagação de buscas por todos os vizinhos de um usuário até um dado raio de profundidade. Esse raio de profundidade é medido por um campo na busca chamado TTL (*time-to-live*) que vai decrescendo a cada pulo da busca para um novo vizinho em profundidade até chegar a zero e a busca ser interrompida.

Esse tipo de algoritmo possui alguns problemas. Além de produzir sub-redes dentro da rede, por limitar o alcance de uma busca, a quantidade de informações que são propagadas na rede a torna pouco escalável. À medida que aumenta o número de usuários, a rede fica muito congestionada por buscas propagadas pelos seus participantes.

Segundo [5], são propostas novas metodologias de busca para minimizar o problema, as quais veremos a seguir.

```
ttl = 1;          /* campo de tempo de vida da busca. */
limite = 10;   /* valor máximo para ttl. */
encontrou = falso;
Enquanto não encontrar o arquivo e ttl <= limite faça
        Para cada vizinho i do nó faça
                encontrou = repassaBusca(i,ttl-1);
        Fim Para;
        Se não encontrou, então
                ttl = ttl +1;
        Fim Se;
Fim Enquanto;
Fim.
```

Figura 6: Pseudocódigo da busca usando algoritmo Expanding Ring

3.2 Algoritmo Expandig Ring

Neste algoritmo, em vez de fazer uma busca com TTL fixo, é utilizada uma busca que vai incrementando o TTL, a partir de um valor baixo, até encontrar o dado desejado ou até que seja atingido o TTL máximo. Essa abordagem possibilita reduzir o tráfego na rede, porém pode aumentar o tempo de duração de uma busca. Por esses problemas o uso desse algoritmo não é aprofundado e os estudos caminham para outras soluções. Um pseudocódigo para este tipo de busca pode ser observado na Figura 6.

3.3 Algoritmo Random Walks

Esse método utiliza-se de buscas apenas em profundidade, escolhendo apenas um vizinho aleatóriamente. Essa abordagem apresenta um atraso em relação ao método *Expanding Ring* por deixar de consultar muitos vizinhos.

Numa tentativa de evolução, o algoritmo consulta o usuário requisitante a cada pulo da busca, assim obtém uma eficiência maior que o possibilita aumentar o TTL. Outra mudança acrescentada foi a introdução de um identificador para cada busca, de modo a minimizar *loops* no caminho. Quando um nó recebe um identificador repetido ele o propaga por outro caminho, assim evita repetir a mesma consulta anterior.

```
ttl = 30;          /* campo de tempo de vida da busca. */
k = 16; /* número de vizinhos que serão consultados ao mesmo tempo */
encontrou = falso;
listaVizinhos = escolhe(k, listaTotal);
Para cada elemento i de listaVizinhos faça
        encontrou = repassaBusca(i,ttl-1);
Fim Para;
Fim Enquanto;
Fim.
```

Figura 7: Pseudocódigo da busca usando algoritmo Random Walks

Outra modificação introduzida foi a propagação da informação por k vizinhos ao mesmo tempo, para diminuir o atraso e ainda assim aproveitar a busca em profundidade desse

método. Porém, o valor de k deve ser bem escolhido para evitar congestionar muito a rede como no primeiro método apresentado. Um valor variando de 16 até 64 para k pode obter um melhor custo-benefício. Um pseudocódigo para este tipo de busca pode ser observado na Figura 7.

3.4 Mecanismo de Busca por Afinidades

Para este projeto foi estudado um mecanismo a ser implementado em combinação com o algoritmo *expanding ring* definido acima para que a cada busca bem sucedida, o parceiro através do qual a busca obteve sucesso será adicionado a uma lista de amigos do usuário.

A idéia por trás desse mecanismo é que a chance de encontramos um arquivo em um "amigo" é muito maior, pois supostamente nossos amigos têm os mesmos gostos que nós. Por isso esse mecanismo é uma busca por afinidades. Antes de tentarmos buscar algum conteúdo por toda a rede, vamos primeiro tentar localizá-lo nas máquinas de nossos amigos.

É claro que nem sempre o usuário conseguirá atingir seu objetivo, porém supõe-se que, caso o nó que retornou o pedido tenha um gosto parecido com o dele, a probabilidade de uma busca bem sucedida aumentará, em comparação com outros usuários com gostos completamente diferentes.

No próximo capítulo serão detalhados o algoritmo implementado para estre projeto e a técnica utilizada.

CAPÍTULO 4 – IMPLEMENTAÇÃO DO SOFTWARE

4.1 Implementação Inicial

O início do projeto foi uma construção de um aplicativo cliente e outro aplicativo servidor para troca de mensagens P2P.

Os aplicativos foram construídos utilizando linguagem JAVA, o que possibilitou um algorítmo focado apenas no uso da camada de aplicação, ou seja, sem se preocupar em como os dados seriam transmitidos, pois já existem instruções prontas na JAVA API. Assim pude desenvolver as funcionalidades do programa de modo imediato.

Nessa primeira parte do projeto, como descrito acima, existem dos tipos de aplicativo. Abaixo, vou explicar brevemente a funcionalidade dos dois.

O aplicativo cliente, ilustrado na Figura 8, é o aplicativo que roda em cada máquina dos usuários finais, os pontos que se comunicam. Basicamente, ao iniciarmos sua execução, devemos informar o endereço IP da máquina onde estará sendo executado o aplicativo servidor, que gerencia as conexões como na arquitetura de um diretório centralizado. Assim o programa se conecta ao servidor e começa a trocar mensagens com os outros usuários conectados ao mesmo servidor.

Já o aplicativo servidor, ilustrado na Figura 9, é aquele que vai gerenciar todas as conexões e possibilitar a troca de mensagens entre os usuários fazendo um repasse da mensagem enviada por um usuário para todos os outros usuários conectados. Quando o programa servidor inicia sua execução, ele começa a esperar por conexões. Quando uma conexão chega, ele adiciona o usuário à lista de usuários e manda essa lista para todos os usuários conectados para que estes saibam quem está conectado naquele momento. Da mesma forma, quando um usuário se desconecta, o servidor retira seus dados da lista.

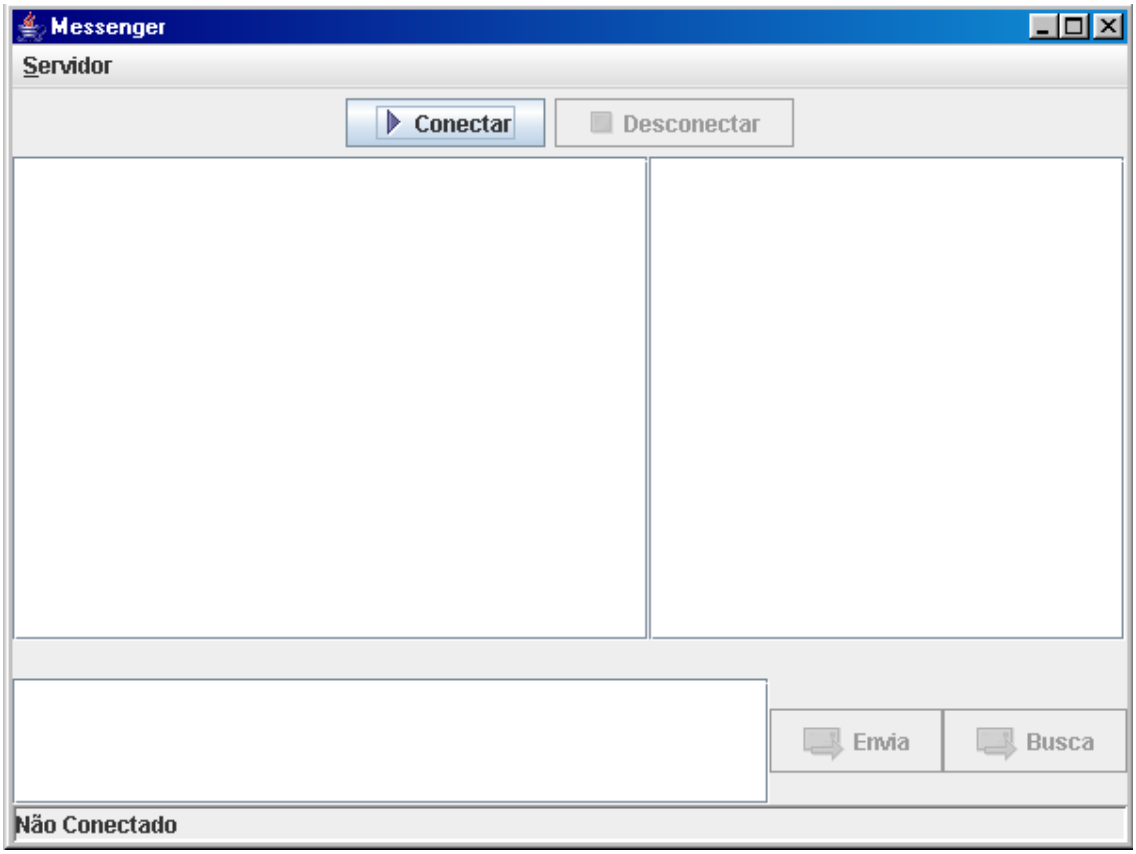

Figura 8: Aplicativo Cliente

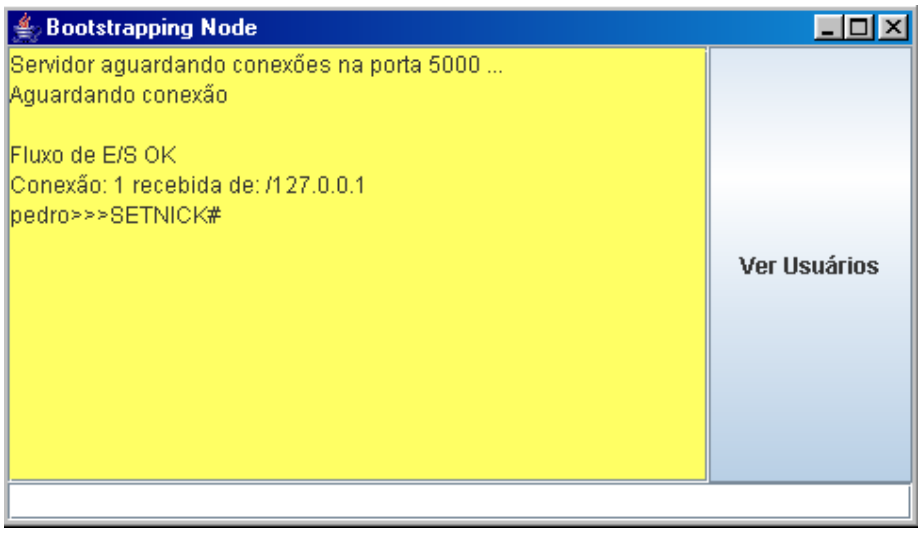

Figura 9: Aplicativo Servidor

Quando o usuário quer se conectar ao servidor, ele executa o programa cliente passando o endereço IP do servidor como parâmetro. Quando a conexão se estabelece, o usuário digita seu apelido (*nickname*) para se juntar ao grupo como numa sala de bate papo.

Esta situação é ilustrada na Figura 10. O apelido escolhido é enviado para o servidor, que envia uma lista contendo todos os usuários conectados a ele, incluindo o do novo usuário.

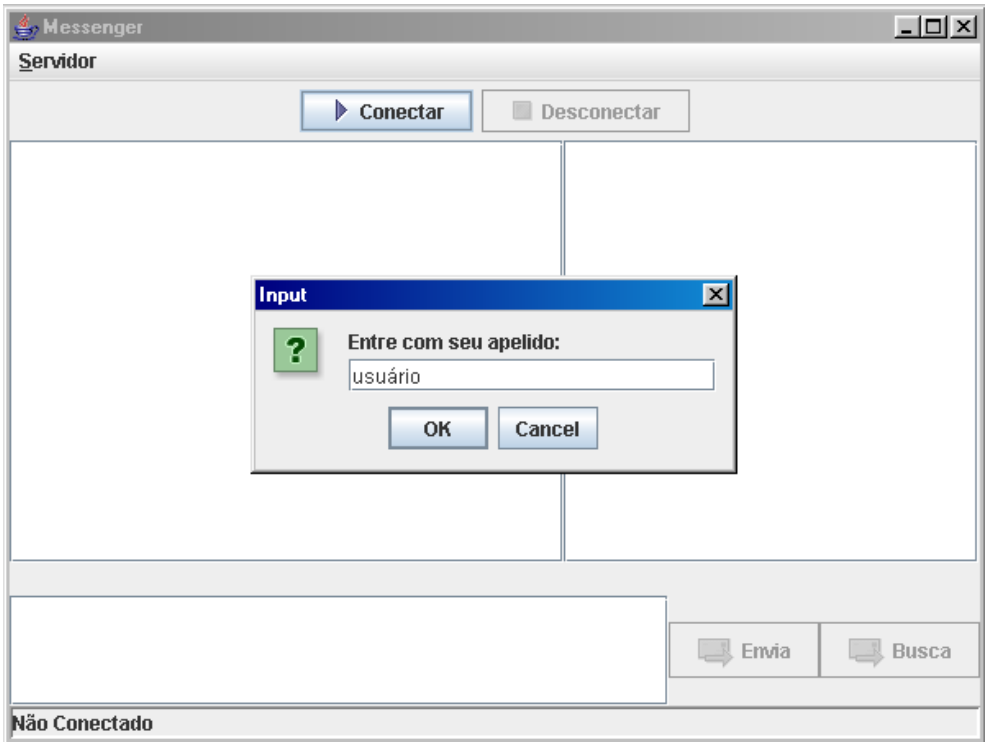

Figura 10: Escolha do apelido

Todos os pacotes enviados para o servidor são tratados como mensagens de texto, que contém o apelido do remetente e o corpo da mensagem propriamenta dita.

Assim, para simplificar o projeto, os comandos especiais, como inicialização do apelido, aviso de conexão e desconecção, etc, são tratados como mensagens "especiais", onde o remetente tem o nome do comando e o corpo da mensagem é o parâmetro passado pelo comando. Por exemplo, quando o cliente envia seu apelido para o servidor, é enviada uma mensagem cujo remetente é um comando reservado e o corpo da mensagem contém o apelido a ser passado (isso é necessário para que o servidor saiba diferenciar uma mensagem de texto que deve ser enviada a todos de um comando passado a ele e que não deve ser enviado aos demais usuários).

Após receber a lista de usuários pode ocorrer a troca de mensagens. Basta o usuário digitar uma mensagem na caixa de texto e pressionar o botão "Enviar".

Quando um outro usuário se junta à lista, os demais recebem uma mensagem de aviso, assim como no caso de uma desconexão. Sempre que algum usuário entra ou sai da lista, ela é

atualizada no servidor e reenviada aos usuários conectados para que estes atualizem suas listas locais (no cliente a lista não existe como estrutura em memória, apenas como texto na interface do programa). Os comandos de limpeza e atualização da lista são reconhecidos pelo aplicativo cliente. O servidor envia um comando para limpar a interface da lista e depois envia mensagens com a lista (um comando para cada apelido). À medida que cada apelido vai sendo recebido, a lista vai crescendo até ficar igual à sua equivalente do servidor.

Nenhum comando se perde devido à utilização do protocolo TCP (que garante o envio) nas trocas de mensagens. Como foi dito anteriormente, as instruções em JAVA permitem a escolha do modelo de transporte, mas ocultam como ele é feito.

O aplicativo cliente funciona com a execução de tarefas (*threads*) para recebimento e envio de mensagens. A tarefa de recebimento fica sempre executando para esperar a chegada de uma eventual mensagem. Quando ela é criada, ela faz a ligação do cliente com o *socket* do servidor, a fim de receber as mensagens enviadas aos demais usuários. Já a tarefa de envio é iniciada quando o usuário pressiona o botão de envio e é destruída após o envio da mensagem. A tarefa de recebimento é finalizada quando o usuário se desconecta do servidor. A troca de mensagens é ilustrada pela Figura 11e pela Figura 12

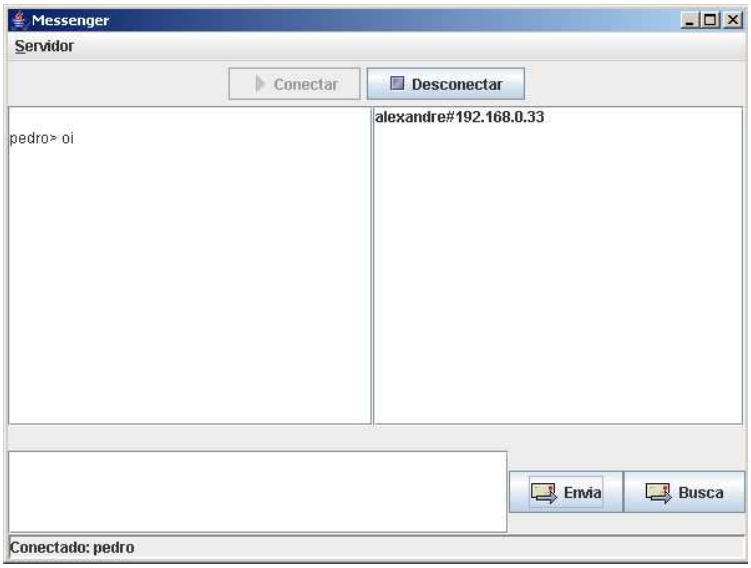

Figura 11: Usuário envia mensagem de texto.

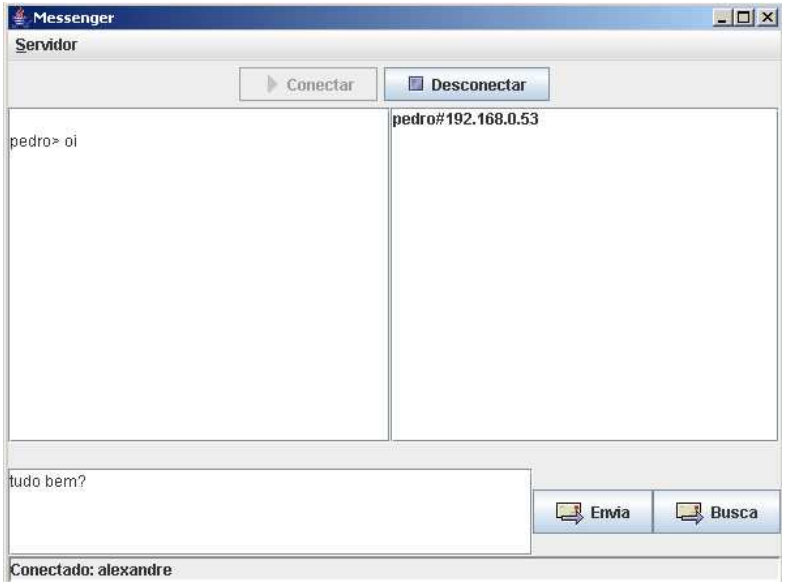

Figura 12: Usuário recebe mensagem e responde.

A função do servidor é gerenciar as conexões de clientes guardando seus apelidos e endereços IP numa lista duplamente encadeada. O servidor também é responsável por enviar essa lista para todos os usuários conectados e realizar a troca de mensagens entre todos os usuários. O funcionamento do servidor é apresentado na Figura 13.

Nesta figura, podemos observar que o servidor aguarda conexões na porta 5000 e recebe conexões de usuários com diferentes endereços IP. São recebidas cinco conexões de diferentes endereços IP. Cada conexão é seguida por uma mensagem com uma palavra reservada *SETNICK* para indicar que o usuário deseja estabelecer seu apelido na rede.

Figura 13: Servidor Gerenciando Conexões

Quando o aplicativo é iniciado, um *socket* servidor é criado para receber as conexões. À medida que os usuários vão se conectando, a lista vai aumentando e o aplicativo envia comandos para limpar a lista, a fim de atualizar os usuários conectados, e depois envia os apelidos um por um até completar a nova lista. Para impedir que o comando de limpar a lista antiga chegue depois de um comando de atualização, assim que é enviada a mensagem de limpeza, a tarefa de envio dorme por 0.5 segundos para então enviar os outros comandos.

Além desses comandos, quando um usuário se conecta ou desconecta, o servidor envia o comando para indicar a ação aos demais.

Para possibilitar o envio de mensagens, como dito acima, o servidor também possui tarefas de recebimento e envio de mensagens. É criada uma tarefa de recebimento para cada usuário que conecta no servidor. Essas tarefas são escalonadas por fatia de tempo para que o servidor possa receber mensagens de todos os usuários. A tarefa de envio envia uma mensagem para o endereço de todos os clientes que estão aguardando mensagens através das suas tarefas de recebimento.

4.2 Busca Radial

A implementação inicial serviu para estudo da comunicação entre clientes e servidores e como realizar troca de mensagens. Em um segundo momento, foi feita a decisão sobre a realização de uma implementação de um mecanismo de busca para redes cooperativas, visto que esse é um assunto que vem sendo tema de vários estudos [5] e, ao mesmo tempo, milhares de pessoas são beneficiadas com buscas mais eficientes.

Esse mecanismo foi implementado tendo como base o *software* desenvolvido inicialmente, pois a estrutura de comunicação já estava pronta e seriam necessárias apenas algumas modificações para realizar uma busca através da rede.

O primeiro *software* construído era centralizado. O servidor repassava todas as mensagens para todos os nós sem que os próprios pudessem se comunicar diretamente. Assim, a primeira modificação feita foi a de transformar o servidor em um nó inicializador, como na rede Gnutella. Todos os usuários passariam a acessar o nó apenas para obter o endereço dos seus vizinhos na rede e a busca seria feita diretamente entre os participantes, sem o controle do nó inicializador.

O nó inicializador é necessário para que a rede nunca seja desfeita. No pior caso, onde todos os usuários estivessem desconectados da rede ao mesmo tempo, a rede não poderia ser re-estabelecida sem a consulta a esse nó.

A aplicação que representa o nó inicializador envia, agora, apenas um número limitado de vizinhos na rede para cada usuário que se conecta para representar mais fielmente a topologia de uma rede descentralizada.

Após essa alteração, o próximo passo foi desenvolver o mecanismo de busca. A aplicação “cliente” recebeu um novo botão de busca para que o texto escrito na caixa de mensagens fosse enviado para os vizinhos como uma mensagem de requisição. Foram criados dois novos comandos reservados “REQUEST”, para representar uma requisição por uma palavra chave, e “RESPONSE” para representar uma resposta de sucesso à mensagem de busca.

Para o tratamento destes tipos de mensagens foi criado um novo módulo de busca na aplicação. Este módulo é iniciado quando o usuário faz a conexão com o nó inicializador e conta com uma tarefa de recebimento própria, para tratar as mensagens a serem recebidas dos outros usuários.

Uma tarefa de envio também foi criada para quando o usuário clicar no botão de busca. Quando isso ocorre, é criada uma tarefa de envio do módulo de busca. Essa tarefa

envia a mensagem de busca para todos os vizinhos que o nó inicializador indicou no início da conexão à rede.

Esse primeiro mecanismo implementado corresponde a uma busca radial simples, pois o usuário envia a busca para todos os seus vizinhos, que por sua vez repassam a mensagem para todos os vizinhos. Porém a busca é limitada pelo raio estabelecido por um campo de tempo de vida que é transmitido junto com a mensagem e decrementado a cada pulo da mesma pela rede. Quando o tempo de vida chega a zero, a busca é interrompida. Como visto anteriormente, essa abordagem cria sub-redes dentro da rede, pois a mensagem de busca tem seu alcance delimitado pelo raio gerado pelo tempo de vida estabelecido.

Quando uma mensagem de busca é recebida, o nó faz uma simulação de busca em arquivo texto e caso encontre a palavra requisitada retorna para o nó inicial, caso contrário repassa a busca para todos os vizinhos se o tempo de vida não for igual a zero decrementando o mesmo em uma unidade.

O objetivo desta implementação é mostrar um mecanismo de busca simples e que pode ser alvo de uma melhoria de desempenho com a introdução de novas idéias.

4.3 Busca por Afinidades

O próximo passo da implementação foi incluir uma melhoria na idéia do primeiro algoritmo de busca radial simples.

O objetivo deste segundo mecanismo é tornar a busca mais eficiente que a anterior pela adição de uma lista de amigos através da qual cada usuário possa consultar antes de recorrer à lista de vizinhos normais. Assim pode-se ganhar tempo admitindo que um amigo tenha os mesmos gostos e a chance de sucesso tende a ser maior.

Essa implementação foi realizada com a adição da uma lista de amigos no módulo de busca. A cada requisição criada ou repassada, a lista de amigos, caso exista, é sempre acionada antes dos vizinhos do nó. Caso contrário, a busca funciona como uma busca radial simples.

Na Figura 14, pode-se observar o diagrama de classes da aplicação.

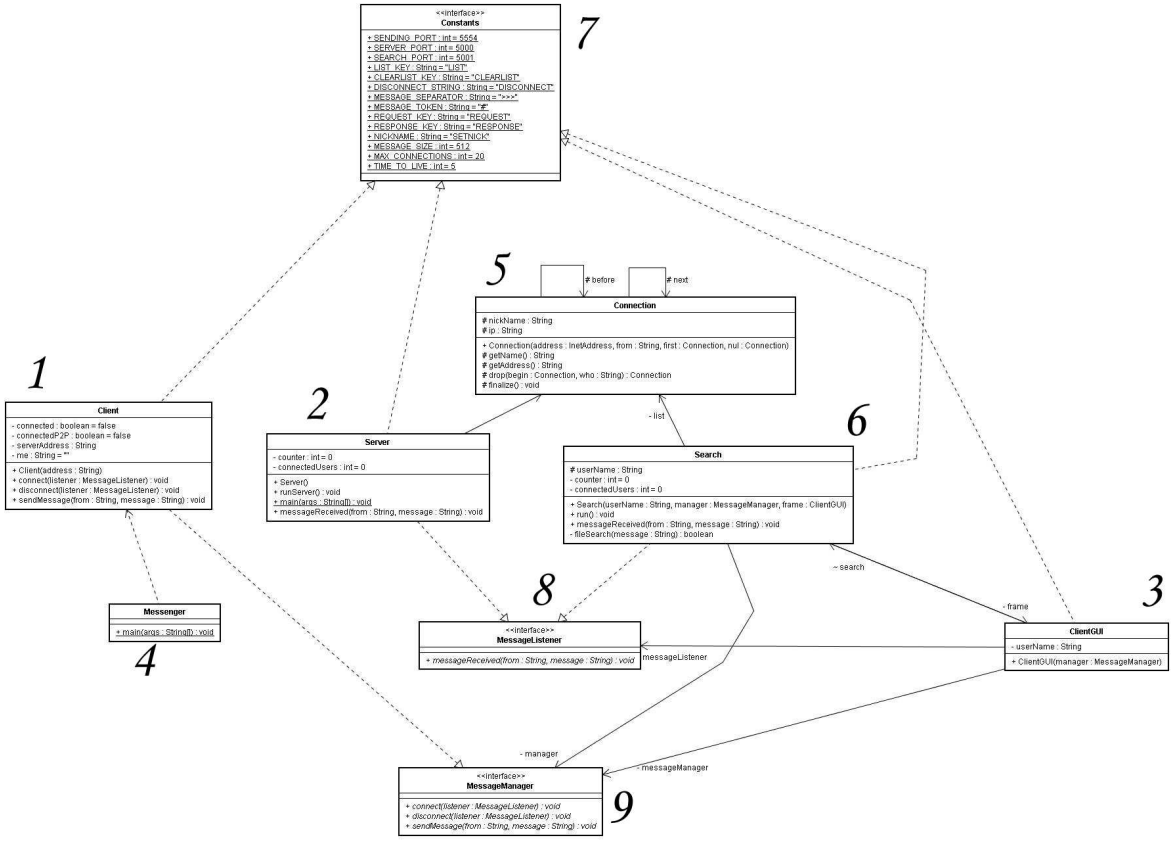

Figura 14: Diagrama de classes da aplicação

Abaixo seguem as classes e interfaces contidas no diagrama da Figura 14.

A classe Client (1) contém todas as funcionalidades relacionadas ao cliente, como tarefas de envio e recebimento de mensagens de texto.

A classe Server (2) contém todas as funcionalidades relacionadas ao nó inicializador, como gerenciamento das conexões recebidas e envio da lista de vizinhos para cada cliente que se conecta.

A classe ClientGUI (3) é responsável pela interface gráfica da aplicação cliente e por fazer a diferenciação entre o envio de uma mensagem de texto ou de busca.

A classe Messenger (4) inicializa a aplicação cliente, criando objetos das classes Client e ClientGUI.

A classe Connection (5) representa um objeto cliente para ser armazenado em uma lista de usuários conectados pelo nó inicializador. Cada objeto de Connection contém um apelido e um endereço IP, assim como funcionalidades para ligar-se à uma lista duplamente encadeada e desligar-se dessa lista.

A classe Search (6) é responsável pela conexão entre clientes, possuindo a inteligência para envio e recebimento de mensagens de busca, e também realizando uma simulação de busca em um arquivo texto na máquina do cliente.

A interface Constants (7) possui todas as constantes utilizadas pelas demais classes (porta de conexão, palavras reservadas, etc), logo ela é implementada pela maioria delas.

A interface MessageListener (8) possui um método para tratar o recebimento de mensagens e é implementada por classes que podem tratar mensagens recebidas (Server, Search e ClientGUI).

A interface MessageManager (9) possui métodos para tratar da conexão e desconexão dos clientes ao nó inicializador e é implementada pela classe Client.

CAPÍTULO 5 – ANÁLISE DE DESEMPENHO

Para afirmar a eficácia do estudo do mecanismo de busca por afinidades são necessárias algumas medições para efeito comparativo entre a busca radial simples e a busca radial com afinidades.

Infelizmente, encontrei alguns problemas para estabelecer a infra-estrutura para um ambiente de testes aceitável. Todos os testes foram realizados em uma rede local.

Assim, foi impossível atestar a melhora no desempenho para a implementação da busca por afinidades, pois o tempo de resposta sempre foi muito reduzido.

Logo, a fim de não deixar este trabalho sem nenhuma reflexão, resolvi fazer uma análise teórica do desempenho esperado para os mecanismos implementados anteriormente.

Essa análise foi baseada no conceito de palavras chaves fortes e fracas. Uma palavra chave forte é uma palavra chave que identifica com grande exatidão e exclusividade o conteúdo desejado. Já uma palavra chave fraca não permite apontar nenhum conteúdo exclusivamente, porém sua busca atinge um grande número de conteúdos.

Espera-se que o mecanismo de busca por afinidades funcione bem quando a busca é feita por palavras chaves fortes, pois apenas com palavras chaves fortes podemos identificar melhor a preferência de cada pessoa.

Como exemplo, pode-se citar uma busca por nome de um autor em comparação à uma busca pela palavra chave fraca “amor”. A busca por autor nos dá uma certeza muito maior de que o “amigo” em questão gosta do que o autor escreve ou do gênero da escrita. Ao passo que o fato de um usuário possuir algum conteúdo com a palavra “amor” não nos permite afirmar que o gosto dele é o mesmo que o nosso.

CAPÍTULO 6 – CONCLUSÃO

Como vimos anteriormente, a busca em redes P2P vem sendo constantemente evoluída. Este trabalho buscou dar mais uma contribuição para tornar as aplicações P2P cada vez mais utilizadas e úteis para todos.

Esta implementação foi de grande valia para o entendimento da comunicação cliente/servidor e P2P.

Outro ponto a ser observado é o incentivo às redes cooperativas não estruturadas e da possibilidade do surgimento de novas tecnologias que aproveitem este e outros estudos para que as pessoas que se conectam às redes cooperativas possam encontrar conteúdos que desejam mais facilmente sabendo onde a chance de encontra-los é maior.

Como trabalho futuro, sugiro que seja implementada uma busca do tipo "*random walk*" utilizando os conceitos de busca por afinidades. Assim pode-se aumentar muito a escalabilidade da rede reduzindo-se a quantidade de mensagens trafegando ao mesmo tempo.

Outra sugestão é utilizar os conceitos deste projeto em uma busca para redes parcialmente centralizadas, visto que são mais eficientes do que as arquiteturas totalmente descentralizadas.

Para que a busca seja mais eficiente, também é necessário eliminar repetições de mensagens que trafegam entre os usuários. Ao adicionarmos um identificador único para a mensagem que é criada, podemos guardá-lo na primeira passagem da mensagem pelo nó e descartar repetições futuras através de uma comparação simples.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Deitel, H. M. e Deitel, P. J., "**Java Como Programar**", Volume 1, Quarta Edição, 2002.

[2] **Sun Microsystems**. http://java.sun.com/, novembro de 2006.

[3] Stephanos Androutsellis-Theotokis e Diomidis Spinellis, "**A Survey of Peer-to-Peer Content Distribution Technologies"**, ACM computing surveys, No. 4, dezembro de 2004.

[4] Siu Man Lui e Sai Ho Kwok, "**Interoperability of Peer-To-Peer File Sharing Protocols"**, ACM SIGecom Exchanges, Vol. 3, No. 3, agosto de 2002.

[5] Qin Lv, Kai Li, Pei Cão, Edith Cohen e Scott Shenker, "**Search and Replication in Unstructured Peer-to-Peer Networks"**, junho de 2002.

[6] Gnutella.com. http://www.gnutella.com/, novembro de 2006.

[7] Kazaa.com http://www.kazaa.com/, dezembro de 2006.

APÊNDICE

O código contido neste apêndice é apenas a parte referente ao módulo de busca da aplicação, implementando a busca por afinidades.

Código do módulo de busca implementado no arquivo Search.java:

```
import java.io.BufferedReader;
import java.io.EOFException;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.InterruptedIOException;
import java.io.PrintWriter;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.StringTokenizer;


public class Search extends Thread implements MessageListener, Constants {
   private MessageManager manager;
   protected String userName;
   private int counter = 0;
   private Connection list;
   private int connectedUsers = 0;
   LinkedList socketList;
   private ClientGUI frame = null;
   public ArrayList friendList = null;
            /**
             *
             */
            public Search(String userName, MessageManager manager, ClientGUI frame) {
                        super();
                        this.manager = manager;
                        this.userName = userName;
                        this.frame = frame;
            }

              public void run()
               {
                        try{
                           // etapa 1: cria um serverSocket.
                            ServerSocket serverSocket = serverSocket = new ServerSocket( SEARCH_PORT,
                            MAX_CONNECTIONS );

                            socketList = new LinkedList();
```

```
            while ( true ){
                    ++counter;
                    Socket peerSocket = serverSocket.accept();
                    socketList.add(peerSocket);
                    new ReceivingThread( this, peerSocket, userName, socketList).start();

                    list = new Connection(peerSocket.getInetAddress(),"usuário "+counter,list,null);
                    connectedUsers++;
            } // fim do while
        } // fim do try
        catch ( EOFException eofException ){
                    System.out.println( "Clientes fecharam a conexão" );
        }
        catch ( IOException ioException ){
                    ioException.printStackTrace();
        }
    } // fim de runServer

    public void messageReceived( String from, String message ){
        String completeMessage = from + MESSAGE_SEPARATOR + message;
        for (int i=0; i<socketList.size();i++){
                    new SendingThread( completeMessage, (Socket) socketList.get(i) ).start();
        }
    }

    public class ReceivingThread extends Thread implements Constants{
        private BufferedReader input;
        private String userName;
        private boolean keepListening = true;
        private MessageListener messageListener;
        private InetAddress host;
        private Socket sendSocket;
        private LinkedList connectedUsers;
        private LinkedList neighbors = new LinkedList();
        public ReceivingThread(MessageListener listener, Socket peerSocket, String userName,
        LinkedList connectedUsers) {
            super("Thread de recebimento: "+peerSocket);
```

```
    try{

        messageListener = listener;

            this.userName = userName;

            this.connectedUsers = connectedUsers;

            sendSocket = peerSocket;

            peerSocket.setSoTimeout(SEARCH_PORT);

            // configura o fluxo de entrada para objetos

            host = peerSocket.getInetAddress();

            input = new BufferedReader( new InputStreamReader( peerSocket.getInputStream() )
            );


    }
    catch(IOException ioException){

            ioException.printStackTrace();
    }

}

public void run(){

   String message = "";
   boolean found = false;

   while (keepListening){

      try{

            message = input.readLine();

            if (message != null) {

                StringTokenizer tokenizer = new StringTokenizer(message,
                MESSAGE_SEPARATOR );

                if (tokenizer.countTokens() == 2){

                    String user = tokenizer.nextToken();
                    String messageBody = tokenizer.nextToken();

                    // Se a mensagem não é de resposta à uma busca.
                    if (messageBody.startsWith(REQUEST_KEY+MESSAGE_TOKEN)){

                        // procura arquivo nesta máquina, pois a mensagem é de request.
                        found = fileSearch(messageBody);


                        if (found){ // Se achou, envia a resposta para quem chamou.

                            StringTokenizer tokenizer2 = new StringTokenizer(messageBody,
                            MESSAGE_TOKEN);
```

```
tokenizer2.nextToken();

int TTL = Integer.parseInt(tokenizer2.nextToken());

StringTokenizer tokenizer3 = new StringTokenizer(user,
MESSAGE_TOKEN);

tokenizer3.nextToken();

String ip = tokenizer3.nextToken();

Socket response = new
Socket(InetAddress.getByName(ip),SEARCH_PORT);

String completeMessage = this.userName + MESSAGE_SEPARATOR +
RESPONSE_KEY + MESSAGE_TOKEN +
TTL + MESSAGE_TOKEN + tokenizer2.nextToken();

new SendingThread( completeMessage, response ).start();


}else{ // Se não achou, propaga a busca caso TTL seja maior que zero.


StringTokenizer tokenizer2 = new StringTokenizer(messageBody,
MESSAGE_TOKEN);

tokenizer2.nextToken();

int TTL = Integer.parseInt(tokenizer2.nextToken());

if (TTL>0){
// propaga a busca diminuindo o TTL.

String searchString = tokenizer2.nextToken();

if (friendList!=null){ // inicia busca pela lista de amigos.

ArrayList auxList = friendList;

for (int i = 0; i < auxList.size();i++){ // procura na lista de amigos

StringTokenizer token = new
StringTokenizer((String)auxList.get(i),MESSAGE_TOKEN);

token.nextToken(); // nickname

Socket neighbor = new
Socket(InetAddress.getByName(token.nextToken()),SEARCH_PORT);

if (neighbor.isBound()){

new SendingThread(user + MESSAGE_SEPARATOR +
REQUEST_KEY + MESSAGE_TOKEN + (TTL-1) +
MESSAGE_TOKEN + searchString,neighbor).start();

}

}
```

```
                    }

                    for (int i = 0; i<frame.neighbors.size();i++){ // procura na lista dos outros usários


                      StringTokenizer token = new
                      StringTokenizer((String)frame.neighbors.get(i),MESSAGE_TOKEN);

                      String neighborNick = token.nextToken(); // nickname
                      String neighborIP = token.nextToken();
                      Socket neighbor = new Socket (InetAddress.getByName(neighborIP),
                      SEARCH_PORT);

                      if (neighbor.isBound()){

                        new SendingThread(user+ MESSAGE_SEPARATOR+
                        REQUEST_KEY+MESSAGE_TOKEN+
                       (TTL-1)+MESSAGE_TOKEN+searchString ,neighbor ).start();

                      }
                    }
            }


        }       // fim do else
}
// A mensagem é de resposta à uma busca bem sucedida.
else{

    StringTokenizer tokenizer2 = new StringTokenizer(messageBody, MESSAGE_TOKEN);

    tokenizer2.nextToken(); // RESPONSE_KEY

    int TTL = Integer.parseInt(tokenizer2.nextToken());

   frame.messageArea.append("\n"+tokenizer2.nextToken()+" encontrado em "+user);

   // Adiciona usuário na lista de amigos.
   if (friendList==null){
                // Se a lista é vazia, cria e adiciona usuário no qual a palavra foi localizada.
        friendList = new ArrayList();
        friendList.add(user);
   }else{

     boolean foundUser = false; // Variável para indicar se encontrou o usuário já incluído na lista para não
//duplicar.
     for (int i=0; i< friendList.size();i++){

        String j = (String) friendList.get(i);

        if (user.equals(j)){ // Compara cada usuário presente na lista com o que está para ser incluído.

        foundUser = true;

     }

    }
   // Se o usuário que possui o arquivo não está na lista de amigos, inclui na lista.
   if (!foundUser){
       friendList.add(user)
```

```
    }


 }

} // fim do else

 keepListening = false; // já tratou a mensagem e pode sair.
}  // fim do if (message != null )

}
catch(InterruptedIOException interruptedIOException){

   continue;  // continua para a próxima iteração para esperar mais

}
catch(IOException ioException){

ioException.printStackTrace();
break;
} // fim do try/catch

}   // fim do while

// fecha a conexão de um cliente


try{
    input.close();

}
catch(IOException ioException){
            ioException.printStackTrace();
}

}  // fim do método run

} // fim da classe Receiving Thread


public static class SendingThread extends Thread implements Constants {


private String messageToSend="";
private Socket sendSocket;


public SendingThread(String message,Socket address){

super("SendingThread");

messageToSend = message;

sendSocket = address;
} // fim do construtor

public void run(){
            try{

                synchronized(sendSocket){
```

```
  PrintWriter writer = new PrintWriter( sendSocket.getOutputStream() );
  writer.println( messageToSend );
  writer.flush();
}

}
catch(UnknownHostException unknownHostException){

  System.out.println("\nNão foi possível encontrar o host "+sendSocket.toString());
}
 catch ( IOException ioException ){

  ioException.printStackTrace();
 }

        }  // fim do método run


 } // fim da classe SendingThread


private boolean fileSearch(String message){

        // faz a busca da palavra chave dentro de um arquivo texto.
StringTokenizer tokenizer = new StringTokenizer(message, MESSAGE_TOKEN);

tokenizer.nextToken(); // REQUEST_KEY
tokenizer.nextToken(); // TTL

String request = tokenizer.nextToken();

File searchFile = new File("search.txt");

try {

        FileReader reader = new FileReader(searchFile);

        BufferedReader bufReader = new BufferedReader(reader);

        String input = "";

        while (input!=null){

        input = bufReader.readLine();

        if (input!=null){
        if (String.valueOf(input.toUpperCase()).contains(request.toUpperCase())){

                return true;

        }
        }
        }


        reader.close();

        } catch (FileNotFoundException e) {
```

```
                e.printStackTrace();
            } catch (IOException e) {

                e.printStackTrace();
            }

            return false;
        }

}
```