Um Sistema de Cache com Tabelas Hash Distribuídas para Aplicações Peer-to-peer

Cecília de A. Castro César, Édipo Crispim A. Souza

Divisão de Ciência da Computação – Instituto Tecnológico de Aeronáutica (ITA)
São José dos Campos – SP – Brazil

cecília@ita.br, edipo.crispim@aluno.ita.br

Abstract. *Implementations of Distributed Hash Table (DHT) used in Peer-to-Peer applications does not include characteristics of the application or the environment to which they serve. As a result of this generality, further messages are required in the initial stage of finding a peer that has content of interest. A more efficient cache system allow to find the desired peer with fewer attempts, and may even find it on the first try in a steady state network. The implementation of the proposed solution for BitTorrent caused a decrease in the number of messages exchanged by peers, thus saving resources and increasing success rate.*

Resumo. *As implementações de Distributed Hash Table (DHT) utilizadas em aplicações Peer-to-peer não contemplam características próprias da aplicação ou do ambiente aos quais elas servem. Como consequência desta generalidade, são necessárias mais mensagens trocadas na fase inicial para encontrar um determinado parceiro que possua o conteúdo de interesse. Um sistema de cache mais eficiente permite encontrar o parceiro desejado com um menor número de tentativas, podendo encontrá-lo até mesmo na primeira tentativa em uma rede estável. A implementação da solução proposta para BitTorrent, ocasionou a diminuição do número de mensagens trocadas pelos parceiros, gerando assim uma economia de recursos e aumento da taxa de sucesso.*

1. Introdução

As aplicações de compartilhamento descentralizado de arquivos, conhecido como Peer-to-peer (P2P) e as aplicações de vídeo de Tempo Real são hoje as aplicações de maior tráfego na Internet [Sandvine, 2012]. Devido a esta intensa utilização, melhorias que acelerem o compartilhamento de arquivos tem grande impacto para os clientes e para a Internet como um todo.

O protocol BitTorrent (BTP – *BitTorrent Protocol*), protocolo mais usado para compartilhamento de arquivos [Cohem, 2008], mantém um componente centralizado, chamado *tracker*, que pode ser um gargalo no processo de descoberta de parceiros com os quais o sistema de compartilhamento deve interagir. O *tracker* é um servidor que contém a lista de todos os parceiros que estão fazendo *download* ou *upload* do conteúdo.

A técnica de DHT (*Distributed Hash Table*) visa tornar o processo de compartilhamento totalmente distribuído, espalhando aleatoriamente os *trackers* entre os nós ativos da rede. A dificuldade de descentralização refere-se a como descobrir quem tem a lista dos parceiros que têm o conteúdo desejado, ou, onde está o *tracker*. Aos *trackers* é atribuída a responsabilidade por chaves, identificações únicas de determinado conteúdo. Implementações usando DHT têm mostrado sua eficácia, pois mesmo com um grande número de nós na rede, ou grande movimentação referente a nós que entram e saem, o desempenho tem superado as implementações tradicionais que usam o BTP. Tolerância a falhas, escalabilidade e desempenho tem motivado as pesquisas nesta área.

Em diversas propostas usando DHT, constrói-se uma tabela em cada nó onde se armazenam as chaves sobre responsabilidade do nó. No sistema proposto aqui, chamado Kabuto, esta ideia é estendida, armazenando não só as chaves de sua responsabilidade, como também as chaves de seu interesse para repetição posterior do mesmo acesso. Dependendo do perfil da aplicação ou da arquitetura do ambiente alvo, manter a informação já utilizada pode ser compensador.

Os diversos algoritmos que implementam abordagens de DHT não dão ênfase a pesquisas que se repetem, seja pela busca da mesma chave novamente, seja pela necessária comunicação com o *tracker*. Nos sistemas de compartilhamento, de tempos em tempos, acontece a atualização da situação de *downloads/uploads* no *tracker*. Mesmo usando DHT, onde os *trackers* estão espalhados aleatoriamente, o *tracker* responsável por determinada chave será repetidamente acessado pelo mesmo nó para completar a operação de *download* ou *upload* ou para efeitos de manutenção.

Visando dar maior eficiência ao processo de comunicação de parceiros, o Kabuto constrói uma nova estrutura de cache que armazena as chaves pesquisadas e informações relevantes. Este tipo de cache pode apoiar qualquer dos diferentes algoritmos de DHT, pois registra os melhores resultados, ou seja, os nós já conhecidos que estão mais próximos do alvo desejado acarretando um número menor de saltos até o alvo.

Na associação do Kabuto com qualquer algoritmo base de DHT, o pior caso da associação coincide com o pior caso do algoritmo base, que é quando o cache ainda não está construído. O melhor caso é de um único salto, quando já se sabe exatamente qual é o *tracker*.

Para verificação de melhoria com o uso deste cache, foi escolhida a implementação de DHT do conhecido algoritmo Kademlia [Maynounkov e Mazieres, 2002], um dos algoritmos amplamente disponíveis em ferramentas de simulação.

Os resultados das simulações indicaram um ganho significativo do desempenho comparando cenários com uso exclusivo do Kademlia e cenários com o Kabuto.

2. Trabalhos Relacionados

O algoritmo Chord foi um dos pioneiros entre as propostas de DHT e muitas das ideias-base nesta linha de pesquisa foram extraídas do Chord [Stoica, 2001]. Este algoritmo propôs dois tipos de identificadores: Identificadores do conteúdo a disponibilizar (Chave) e Identificadores dos nós (*NodeId*). Quanto à construção destes identificadores:

- Chave: de maneira similar ao BitTorrent, a descrição do arquivo que se deseja baixar está em um meta-arquivo. A partir deste meta-arquivo, o Chord calcula uma chave que o identifica unicamente no anel, através da aplicação de uma função hash: Chave=Hash(descrição do arquivo);
- *NodeId*: o Chord mapeia o endereço IP dos nós em identificadores, também usando a função hash: *NodeId*=Hash(IP).

Após obter estes identificadores, atribui-se a um dos nós a responsabilidade pela chave, ou seja, este nó será o *tracker* para o arquivo descrito pela chave. Em um sistema com N nós, o Chord mantém informação sobre outros *O*(log N) nós, e encontra o *tracker* via *O*(log N) mensagens para outros nós.

Vários algoritmos se seguiram ao Chord na tentativa de melhorar seu desempenho e em todos está presente a ideia seminal de chaves e identificadores de nós. Variam entre as implementações os critérios de atribuição de responsáveis de chave e os algoritmos de busca e recuperação de chave.

Entre as muitas implementações de DHT que se sucederam, estão o Kademlia [Maymounkov e Mazieres, 2002], Tapestry [Zhao, 2004], Epichord [Leong, 2004], a proposta de [Mizrak, 2003] e de [Gupta, 2004].

O Kademlia [Maymounkov e Mazieres, 2002] é um eficiente algoritmo amplamente disseminado e, como vital para o Kabuto, será examinado na próxima seção do artigo.

O Tapestry [Zhao, 2004], também entre os pioneiros, tem um protocolo complicado que dificulta a análise formal de seu desempenho.

O Epichord [Leong, 2004] acrescenta à ideia do Chord, o envio de *p queries* em paralelo para os nós que estão em seu cache, selecionando as melhores respostas e promovendo o crescimento ordenado do cache o que melhora o desempenho médio. O pior caso do Epichord, e não a media, tem desempenho *O*(log N). Embora o Epichord também utilize cache, não tem a preocupação de manter este cache para uso posterior, apagando de modo geral as entradas após certo tempo fixo de uso.

Alguns dos algoritmos tentaram melhorar o desempenho através de estratégias visando escalabilidade, uma vez que, com *O*(log N), uma rede com um milhão de nós necessitaria em média de 20 saltos para localizar o item de interesse, o que significa ainda uma latência muito alta. De fato, o estudo de [Poulwelse, 2005] verificou que o número de downloads de apenas um arquivo popular chegou a 50.000 parceiros, enquanto que o estudo de [Saroiu, 2002] com o Gnutella indicou tipicamente de 20.000 a 40.000 parceiros a qualquer instante. Estes estudos indicam que a rede de overlay formada pode ser de grande dimensão; quanto mais conhecidos e utilizados os programas de distribuição de conteúdo, pior o problema da localização de parceiros.

Algoritmos que consideram a especificidade do ambiente a que se dedicam são vantajosos. Um exemplo disto é a associação de DHT ao protocolo de voz SIP que alia a distribuição de chave às mensagens do protocolo SIP para acelerar o tempo de estabelecimento da chamada [Yu, 2012]. Por não se apoiar em ponto único de falha, DHT associado ao SIP tem despontado como uma solução também indicada para suporte à mobilidade em redes Ad Hoc [Yahiaoui, 2012]. Estes trabalhos apoiam a idéia que é vantajoso considerar o ambiente e comportamento da aplicação.

O problema da localização de parceiros é agravado se os parceiros têm enlaces com diferentes capacidades. [Mizrak, 2003] propõe uma estratégia híbrida que distribui parcialmente a informação para encontrar parceiros. Nós com maior capacidade, chamados super-parceiros, fazem parte ativa do sistema de localização e compartilham o índice com parceiros conectados a eles, agindo como procuradores para os parceiros a eles confiados.

A ideia de super-parceiros parece muito indicada para o ambiente móvel, onde os nós móveis costumam ter menor capacidade que os nós fixos e poderiam ser o gargalo no sistema de localização de parceiros. O sistema Chordella [Hofstatter, 2008] propõe que os super-parceiros, que são os nós fixos, troquem informação de carga para determinar quais dos super-parceiros pode ficar responsável por um novo parceiro móvel que quer entrar, promovendo assim uma distribuição de carga entre os super-parceiros.

A proposta de [Gupta, 2004] dissemina intensamente as informações de trackers entre todos os membros de maneira que, todos os nós com informações precisas de roteamento, com apenas um salto atingem o alvo. Os autores argumentam que até poucos milhões de nós este esquema é compensador. Para uma rede de overlay com mais nós, apresentam um esquema de dois saltos em média, impondo uma quantidade fixa e relativamente alta de tráfego para atingir esta meta.

O algoritmo proposto aqui não acrescenta *queries* novas na rede e não pretende manter informação atualizada de todos os nós na rede. O Kabuto registra as melhores pesquisas feitas em um cache local. Esta heurística tem potencial para ser incorporada em qualquer dos algoritmos expostos.

Uma comparação completa entre os algoritmos deve considerar não só a latência da busca, como também o custo da comunicação, devido ao tráfego de manutenção [Li, 2004].

O desenvolvimento do Kabuto foi integrado ao Kademlia que será brevemente apresentado a seguir.

3. Visão geral do algoritmo Kademlia

O Kademlia [Maymounkov e Mazieres, 2002] é um algoritmo de fácil implementação que está disponível em diversas plataformas de simulação e por isto foi o algoritmo base escolhido para a implementação do Kabuto.

Nesta seção serão apresentadas as principais características do Kademlia necessárias para a compreensão do Kabuto.

Nos algoritmos de DHT, se houver um nó com identificador igual à chave, ele se tornará o *tracker* desta chave. Se não houver, o nó ativo seguinte à chave ficará responsável por ela. A função *sucessor*(chave) aponta para o nó ativo seguinte à chave. O critério de proximidade varia entre os algoritmos. A seguir, será apresentado o critério de proximidade entre nós do Kademlia, sua estrutura de dados e resumo do protocolo.

3.1. Proximidade baseada em XOR

O Kademlia usa uma métrica para calcular a distância entre os nós no espaço de endereçamento baseada em XOR, usando chaves de 160 bits. Para encontrar uma chave, Kademlia calcula a distância entre dois identificadores da seguinte maneira: distância de *x* a *y* é (*x* XOR *y*). Desta maneira a distância é simétrica: d(*x,y*) = d(*y,x*).

O Kademlia dispõe os identificadores dos nós que são números de *m* bits em uma árvore binária. A noção de proximidade entre identificadores (*NodeIds*) na árvore é: uma folha mais próxima de um *NodeId x* é a folha cujo *NodeId* compartilha o maior prefixo comum com *x*. Assim, por exemplo, em uma rede com m=3 bits, o *NodeId*=110 está mais próximo do *NodeId*=111 do que do *NodeId*=101. De fato, d(110,111)=1 enquanto que d(110, 101)=3. A Figura 1 ilustra esta árvore binária com possibilidade para até 8 nós, que, a um dado momento, podem não estar todos ativos.

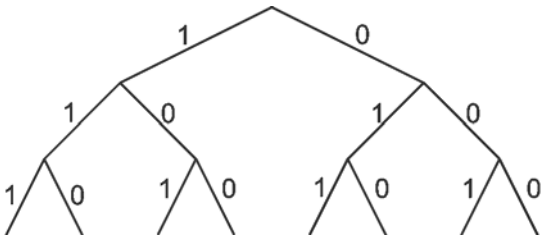

Figura 1 – Árvore de identificadores dos nós para *m*=3 bits.

3.2 Estrutura de dados *k-bucket*

Existe uma estrutura de dados com informações de contato dos nós, as chamadas listas de *k-buckets*. Uma determinada lista contém informação dos nós conhecidos que estão a certo intervalo de distância: a i-ésima lista contém *k* nós no intervalo de distância $[2^i,2^{i+1})$. A informação armazenada associada à chave é uma tripla com *NodeId*, IP e porta de contato. A tabela de roteamento do Kademlia é a árvore binária cujas folhas são os *k-buckets*.

A Tabela 1 ilustra um exemplo de conteúdo desta estrutura no nó com *NodeId*=6. Neste exemplo $0 \leq i < 3$ e k=2. Na rede de exemplo com até 8 nós, se supõe que o nó 3 (011) não está participando da rede e os demais nós são ativos. O nó 7 é o mais próximo do nó 6, os nós 4 e 5 estão à distância 2 e 3 respectivamente e como estão ativos, foram encaixados no 2-bucket associado a i = 1. Os nós 1 e 2 estão à distância 7 e 4 respectivamente. Poderia se supor um cenário onde o nó zero está ativo, mas ficou fora desta tabela. O nó zero está a distância 6 do nó 6, porém como se permitem apenas 2 elementos na lista e os nós 1 e 2 chegaram primeiro, não houve espaço para o nó zero; caso o nó 6 queira se comunicar com o nó zero, deve fazer uma busca a partir dos nós que estão na entrada correspondente a i = 2.

Tabela 1: Exemplo das listas de *k-buckets* no nó 6 para k = 2.

i	**Intervalo de Distância**	**k-bucket**
0	[1,2)	(IP1, Port1, NodeId=7)
1	[2,4)	(IP2, Port2, NodeId=4), (IP3, Port3, NodeId=5)
2	[4,8)	(IP4, Port4, NodeId=1), (IP5, Port5, NodeId=2)

Quando o Kademlia recebe uma mensagem de outro nó, atualiza o *k-bucket* para o *NodeId* do remetente.

3.3 Protocolo

O protocolo do Kademlia consiste de 4 chamadas remotas: *Ping* (verifica se o nó está online), *Store* (armazena uma chave), *Find_Node* (retorna a lista de *k-buckets* próximos ao *NodeId*), *Find_Value* (retorna valor associado a chave se encontrar ou o *k-bucket* de menor distância, de maneira análoga ao *Find_Node*).

O algoritmo para buscar uma chave é o seguinte:

- Escolhe α nós do *k-bucket* não vazio mais próximo;
- Envia *Find_Value* em paralelo para os α nós;
- Envia recursivamente *Find_Value* para os nós que retornaram da busca anterior; Termina quando obteve respostas dos *k* nós mais próximos que encontrou.

α é um parâmetro de todo o sistema de concorrência, sendo possível utilizar estimativa do tempo de viagem, RTT (*Round Trip Time*) para selecionar os α nós.

4. Kabuto

Para atender os requisitos de responsabilidade na rede P2P, o Kademlia cumpre bem seu papel. Todavia é possível explorar os interesses de um nó para melhorar o desempenho. Em determinado nó, o conjunto de chaves de seu interesse é diferente do conjunto de outro nó, e embora este conjunto seja mutável, ele se mantém constante por intervalos de tempo suficientemente grandes para tirar proveito desta informação.

A estrutura de cache proposta, chamada *KeyCache* armazena informações que julga relevantes para futuras ocorrências de uma mesma pesquisa.

A seguir será descrita a estrutura de cache com o algoritmo a ser embutido no DHT e a estrutura modular do software desenvolvido.

4.1 Estrutura de dados *KeyCache*

As informações necessárias estão descritas no seguinte pseudo-código:

```
struct KeyCache {
        Key   key ;                 // chave de interesse
        vector <Node> nodes ; // os melhores nós conhecidos para key
        time   last_seen ;          // valor de relógio da última pesquisa feita sobre key
        int     query_count ;     // número de pesquisas feitas sobre key
} ;
```

Para facilitar a migração entre diferentes soluções DHT, não existem chamadas a funções específicas do Kabuto pela aplicação, sendo ele considerado uma extensão da solução de DHT. Na integração com o Kademlia, a adição do cache ocorre em tempo de execução no processo do *Find_Value*. Dessa forma a declaração das chaves ocorre de forma implícita.

Quando uma chave se mostra de interesse pela primeira vez é criado o *KeyCache* que armazena a chave em questão. O campo *nodes* é inicializado vazio, com potencial para o mesmo número de nós do *k-bucket* do Kademlia. O campo *last_seen* recebe o valor do relógio e *query-count*=1.

A decisão de manter a arquitetura e interfaces entre a aplicação e o Kademlia impede uma chamada explícita para destruir o *KeyCache*, porém o cache não deve ser eterno, deixando de ter sentido se a chave deixa de ter interesse. O critério de remoção de uma chave é inversamente proporcional ao número de pesquisas feitas registradas no *query_count*: quanto mais pesquisas foram feitas, mais interessante é a chave, e, portanto só deixa o cache se ficou por muito tempo sem uso.

Uma vez criada uma chave o desejável seria registrar o nó que é o *tracker*, porém enquanto não se conhece o *tracker*, registram-se os nós mais próximos do *tracker* que provavelmente têm informação mais confiável e atualizada da vizinhança. Sempre que um nó faz contato, o Kademlia tenta associá-lo a um *k-bucket* e também o *KeyCache* verifica como e se deve incluí-lo.

Um nó é adicionado ao campo *nodes* se o *KeyCache* tem espaço, pois no início os melhores nós são os nós conhecidos. Se o *KeyCache* está cheio, o nó deve se mostrar mais valioso que os nós conhecidos até o momento. O nó mais valioso é o que está mais próximo da chave segundo os seguintes critérios: em primeiro lugar o nó que sucede a chave, que talvez já seja o próprio *tracker*; em segundo lugar os nós mais próximos que antecedem a chave, pois no caso de queda do sucessor, estes nós seriam os melhores candidatos para informar o novo sucessor; e por último os sucessores por proximidade.

No decorrer da execução, é possível que alguns nós se retirem da rede por conta própria ou por falhas como queda de energia, ou outro evento sem controle do sistema. Em ambas as situações o *KeyCache* deve refletir a falta de contato com o nó.

No caso de um nó se retirar espontaneamente da rede, ele envia mensagem para todos os nós conhecidos. O Kademlia tem um procedimento para retirar o nó do *k-bucket*, e neste momento o Kabuto também retira o nó se estiver no *KeyCache*. Todas as entradas devem ser varridas, procurando no campo *nodes* o nó que se despede.

No caso de saída não intencional, deve haver uma remodelação de responsabilidades. O sucessor do nó que saiu deve assumir as responsabilidades dele e os interessados no nó que saiu também devem detectar este evento. O Kademlia registra o instante que o nó foi visto pela última vez, e após um tempo máximo faz um *Ping* para verificar se ainda está ativo.

O novo algoritmo de pesquisa busca se beneficiar das informações do *KeyCache*. No processo de pesquisa de uma chave, o mais importante não é mais o *k-bucket*, mas o *KeyCache* associado à chave buscada.

A primeira pesquisa determina o conjunto de nós a ser contactado:

- Usar todos os nós presentes no *KeyCache* associado à chave – potencialmente são k nós;
- Se o conjunto de nós não estiver cheio, usar os nós presentes no *k-bucket* cujo intervalo contém a chave;
- Se o conjunto de nós ainda não estiver cheio, usar os nós presentes no *k-bucket* vizinho do *k-bucket* anterior.

4.2 Estrutura modular do Kabuto

A implementação do Kabuto foi feita no simulador de redes peer-to-peer Oversim [Baumgart, 2007]. Este simulador é um framework com bibliotecas extensíveis e modulares que já possui implementações dos principais protocolos DHT. A Figura 2 mostra a estrutura do Kabuto que define uma classe *KeyRouting* com o código desenvolvido para gerência do *KeyCache* e inclui o código do Kademlia disponível no simulador. Nesta Figura os quadrados pretos representam o *k-buckets* e os quadrados brancos representam o *KeyCache*.

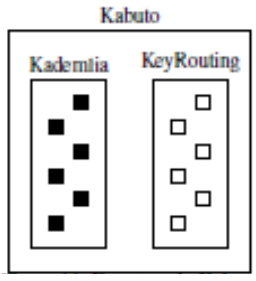

Figura 2: Estrutura do Kabuto em um nó do simulador OverSim

Foi desenvolvida uma aplicação de compartilhamento de arquivos nos moldes do BitTorrent que ativa a implementação de DHT desenvolvida.

5. Testes Realizados

Foram realizados testes aumentando gradativamente o número de nós na rede e comparando o desempenho do Kabuto com o Kademlia e com o Chord. Os gráficos exibidos nesta seção correspondem a cinco simulações de 500 segundos cada. As barras de erro nos gráficos indicam o intervalo de confiança de 95%.

A aplicação de teste desenvolvida no simulador OverSim gerou inicialmente uma rede de parceiros com o número de nós pré-estabelecido. Nestes testes iniciais a rede é estática não contemplando entrada e saída dinâmica de nós, porém foi prevista a perda aleatória de mensagens conferindo certo realismo aos cenários de teste. A aplicação gera aleatoriamente chaves representativas de conteúdos desejados. Cada nó escolhe, também aleatoriamente, uma quantidade de chaves que tem interesse.

Para medir o desempenho foram selecionadas algumas métricas fornecidas pelo OverSim que permitem analisar o volume de dados trocados e o tempo gasto no processo de localização da chave. As métricas são:

- Média de mensagens de pesquisa enviadas por segundo: a cada solicitação de chave, são emitidas mensagens *Find_Value* e as mensagens do protocolo até encontrar a chave desejada;
- Total de mensagens por segundo: incluem–se mensagens para encontrar a chave, mensagens de manutenção emitidas pelos nós e *Pings*;
- Média de pacotes enviados: média de pacotes enviados durante a simulação;
- Latência das mensagens *Find_Value*: tempo gasto no envio de mensagens de busca de chave;
- Latência das mensagens *Store*: tempo gasto nos pedidos de armazenamento de chaves.

5.1 Resultados das simulações

Para os três algoritmos analisados – Chord, Kademlia, Kabuto - a aplicação de compartilhamento que os ativou foi a mesma, e verificou-se que a média de requisições de conteúdo por segundo se manteve constante nos três casos, mesmo aumentando o número de nós na rede. Com o mesmo número de solicitações por parte da aplicação, o Kabuto apresenta a menor média de mensagens de pesquisa, como ilustra a Figura 3. Este resultado reflete que se encontra o nó responsável quase sempre na primeira pesquisa, após certo tempo de construção do cache quando a rede já está estável.

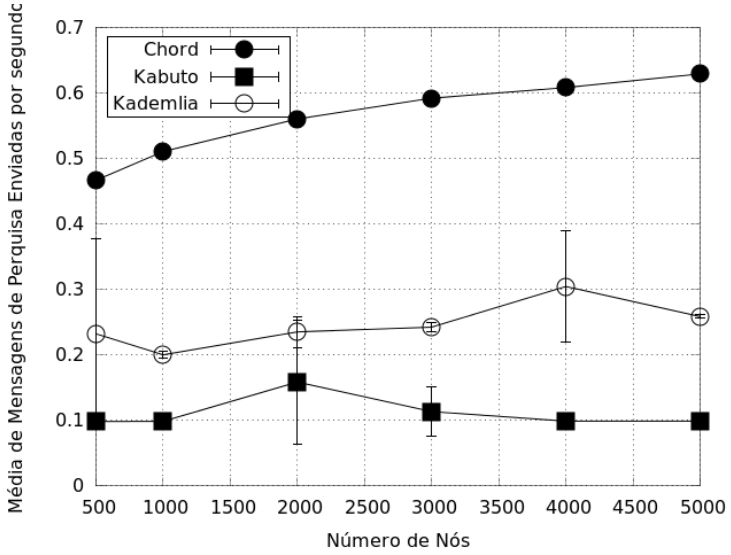

Figura 3 – Mensagens de Pesquisa por Segundo

Considerando o Total de Mensagens enviadas por todo o sistema P2P, também o Kabuto tem bom resultado, como apresentado na Figura 4. O Total de Mensagens apresenta um leve crescimento em relação ao crescimento do número de nós, pois nas redes maiores, há mais nós a serem arguidos pelo gerenciamento. Todavia, este gráfico apresenta que, no total, não há custo adicional no sistema de construção de cache, pelo contrário, o cache auxilia o sistema a enviar menos mensagens, pois se possui informação precisa e atualizada.

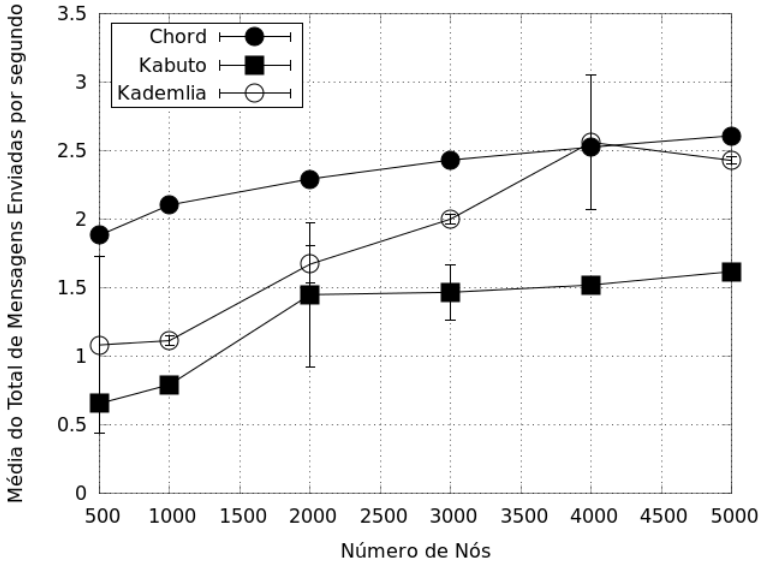

Figura 4 – Total de Mensagens Enviadas por Segundo

Não considerando o volume de dados referente à troca das partes dos arquivos pelos parceiros, mas apenas os pacotes enviados no processo de descoberta do *tracker*, há uma redução deste volume com o Kabuto, como mostra a Figura 5.

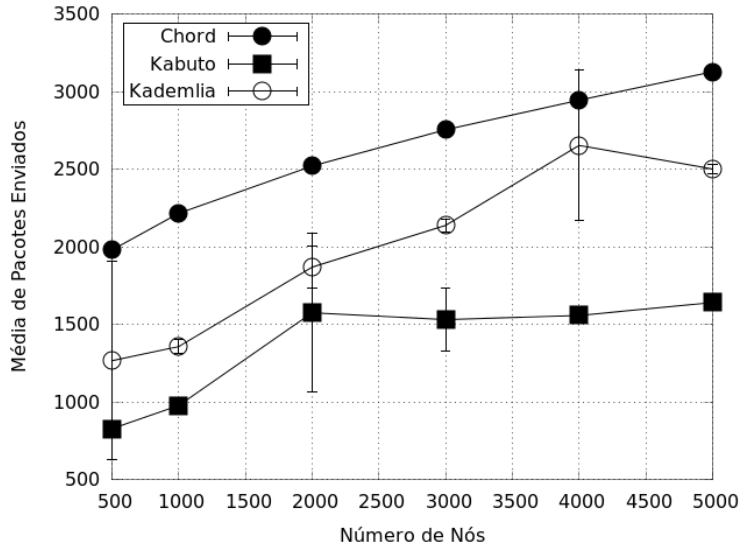

Figura 5 – Total de Pacotes enviados na simulação

Os próximos dois gráficos revelam o ganho em tempo do processo de pesquisa pela chave. Na Figura 6, mostra-se o tempo médio para entrar em contato com o nó responsável por determinada chave, e na Figura 7, o tempo médio para armazenar uma chave.

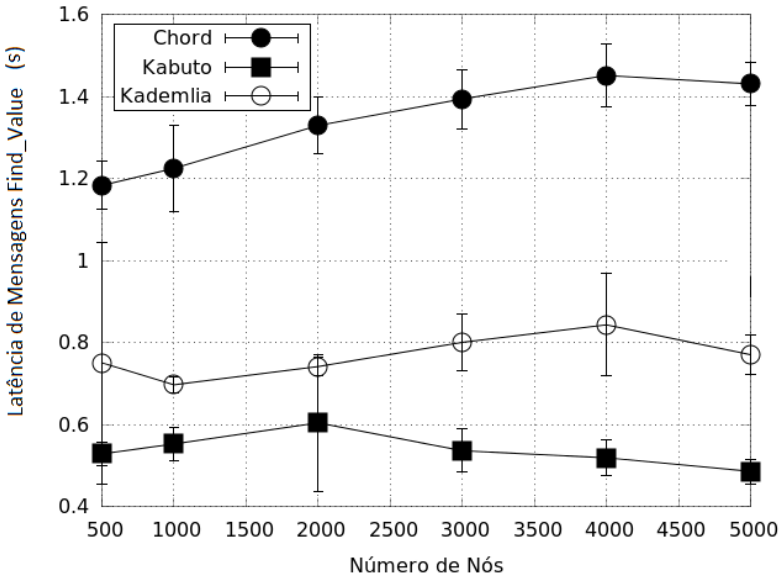

Figura 6 – Latência das Mensagens Find_Value

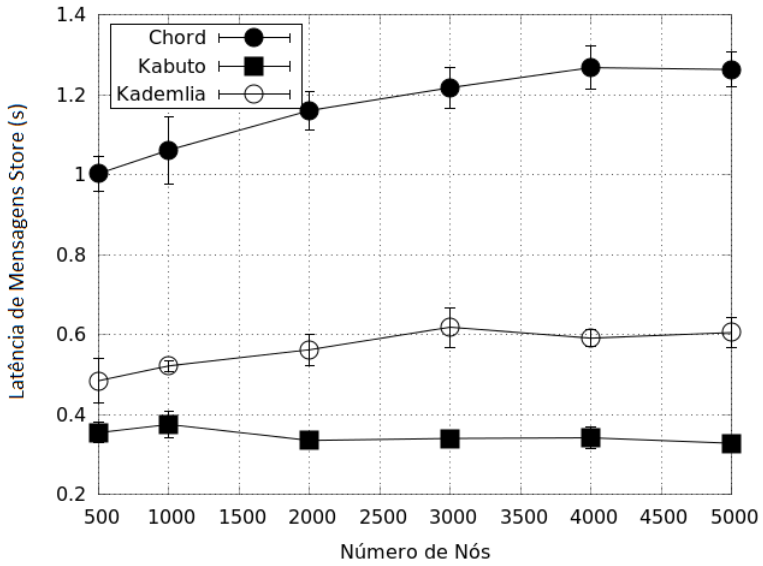

Figura 7 – Latência das Mensagens Store

Destes gráficos, nota-se que o esquema de cache trouxe melhorias tanto na quantidade de banda gasta como no tempo gasto no acesso à chave.

5.2 Consistência de Cache

O Kabuto mantém um controle sobre a permanência da informação no cache. As chaves mais usadas só saem do cache após tempo configurável. Quando um nó deixa a rede espontaneamente, ele avisa os nós conhecidos e neste caso, tanto o algoritmo base, no caso o Kademlia, como o Kabuto, retiram o nó em questão de suas estruturas de dados.

O problema da consistência acontece no caso de saídas não intencionais causadas por eventos sem controle do sistema. Neste caso, o sucessor do nó que saiu deveria assumir suas responsabilidades, mas não o faz imediatamente; haverá uma consulta indevida ao nó que saiu. Ao constatar a ausência de resposta do nó pesquisado, o Kabuto retira o nó do cache e a pesquisa pelo novo nó responsável deve reiniciar pelo algoritmo base.

Esta inconsistência será tratada pelo algoritmo base que quando localizar o novo nó responsável pela chave volta a incorporar esta informação no cache. Desta maneira, no caso de inconsistência, com a utilização do Kabuto haverá apenas uma pesquisa frustrada a mais que o algoritmo base.

6. Conclusão

A solução proposta utilizando uma estrutura de cache adicional facilita a descoberta do responsável por determinada chave. Na comparação feita com o Chord e o Kademlia, algoritmos conhecidos da área, o Kabuto apresenta melhor desempenho, tanto no volume de dados trocados quanto no tempo estimado para encontrar a chave. O Kabuto é de simples implementação, apenas embutido no algoritmo de DHT, porém traz grandes ganhos no resultado final.

A integração deste cache com outras implementações de DHT deve ser verificada. Supõe-se que algoritmos que emitem menos mensagens tenham consequentemente ganho menor. Todavia, espera-se que, mesmo nestes algoritmos, manter em cache informação de consultas precisas e atualizadas deve ser compensador. É preciso investigar o comportamento do Kabuto em uma rede que contemple entrada e saída de nós em um cenário dinâmico.

No futuro há a intenção de desenvolver um arcabouço de simulação que contenha ambiente móvel com a estrutura de super-parceiros, pois há expectativa de ganho no número de mensagens trocadas. Os nós de maior capacidade, super-parceiros, armazenariam pesquisas já feitas para algum parceiro móvel e como ao mesmo tempo são responsáveis por outros parceiros, em caso de repetição da consulta, não emitiriam nenhuma mensagem nova de busca, apenas consultariam o próprio cache com a resposta. Para este estudo, o Kabuto deve ser integrado a algoritmos como o Chordella ou o trabalho de [Mizrak, 2003].

Outra expectativa de melhoria a ser verificada é a utilização da estrutura de cache para aplicações que necessitem de balanceamento de carga. O número de acessos repetidos que é mantido no cache pode ser informação útil para a tomada de decisão sobre a carga a atribuir aos nós da rede.

References

Baumgart, I., Heep, B. and Krause,S. (2007). “OverSim: A flexible overlay network simulation framework”. In Proceedings of 10th IEEE Global Internet Symposium (GI ’07) in conjunction with IEEE INFOCOM 2007, Anchorage, AK, USA, pages 79–84.

Cohen, Bram. (2008) "The BitTorrent Protocol Specification", versão 11031, Fevereiro de 2008 (http://bittorrent.org/beps/bep_0003.html em 30/09/09).

Gupta, A., Liskov, B., Rodrigues, R. (2004). “Efficient routing for peer-to-peer overlays”. in: First Symposium on Networked Systems Design and Implementation (NSDI ´04).

Hofstatter, Q., Zols, S. ; Michel, M. ; Despotovic, Z. ; Kellerer, W. (2008). “Chordella - A Hierarchical Peer-to-Peer Overlay Implementation for Heterogeneous, Mobile Environments”. Proceedings of Eighth International Conference on Peer-to-Peer Computing (P2P '08).

Leong, B., Liskov, B. and Demaine, E. D. (2006). “Epichord: Parallelizing the chord lookup algorithm with reactive routing state management”. Comput. Commun., 29(9):1243–1259. ISSN 0140-3664. (http://dx.doi.org/10.1016/j.comcom.2005.10.002).

Li, J., Stribling, J., Gil, T., Morris, R., Kaashoek, F. (2004). “Comparing the performance of distributed hash tables under churn”, in: The 3rd Int’l Workshop on Peer-to-Peer Systems, February 26–27, 2004.

Maymounkov, P., Mazieres, D. (2002). “Kademlia: A peer-to-peer information system based on the xor metric”, in Proceedings of the 1st International Workshop on Peer-to-peer Systems (IPTPS ´02), Mach, 2002.

Mizrak, A.T.; Yuchung C., Kumar, V., Savage, S. (2003). "Structured superpeers: leveraging heterogeneity to provide constant-time lookup," Internet Applications. WIAPP 2003. Proceedings. The Third IEEE Workshop on , vol., no., pp. 104- 111.

Pouwelse, J., Garbacki, P., Epema, D. and Sips, H. (2005) "The bittorrent P2P file-sharing system: Measurements and analysis", in Proceedings of the International Workshop on Peer-to-peer Systems (IPTPS ´05).

Sandvine. (2012) Global Internet Phenomena Report. http://www.sandvine.com/downloads/documents/Phenomena_2H_2012/Sandvine_Global_Internet_Phenomena_Report_2H_2012.pdf

Saroiu, S., Gummadi, P.K., Gribble, S. D. (2002). “A Measurement Study of Peer-to-Peer File Sharing Systems”. In Proceedings of Multimedia Computing and Networking (MMCN´02).

Stoica, I. , Morris, R., Karger, D., Kaashoek, M. F., Balakrishnan, H. (2001). “Chord: A scalable peer-to-peer lookup service for internet applications” in SIGCOMM Comput. Commun. Rev., 31(4):149–160. ISSN 0146-4833. (http://doi.acm.org/10.1145/964723.383071).

Yahiaoui, S., Belhoul, Y., Nouali-Taboudjemat, N., Kheddouci, H. (2012). "AdSIP: Decentralized SIP for Mobile Ad Hoc Networks," waina, pp.490-495, 2012 26th International Conference on Advanced Information Networking and Applications Workshops.

Yu, L.(2012) "Improving Query for P2P SIP VoIP," Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on , pp.1735-1740. doi: 10.1109/TrustCom.2012.183. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6296193&isnumber=6295938

Zhao, B., Ling, H., Stribling, J., Rhea, S., Joseph, A., Kubiatowicz, J. (2004). “Tapestry: A resilient Global Scale Overlay for Service Deployment”. IEEE J. on Selected Areas in Commun., Vol. 22, p.41-53.