JKad: Middleware para distribuição e descentralização de dados

Bruno C. A. Penteado

Faculdade SENAC de Ciências Exatas e Tecnologia
São Paulo – SP – Brasil

Av Eng. Eusébio Stevaux, 823 – CEP 04696-000

bruno.penteado@gmail.com

Abstract. *Neste artigo é proposta a construção de um middleware entre uma dada aplicação e uma fonte de dados, atuando como um distribuidor de dados de forma totalmente ou parcialmente descentralizada. Tal middleware será baseado na rede ponto a ponto Kademlia, escolhida pela sua relativa simplicidade e por oferecer maior quantidade e qualidade de recursos. Para a validação deste middleware e aspectos vindos da Kademlia como escalabilidade, disponibilidade e robustez, serão produzidos testes automatizados de funcionalidade e performance.*

1. Introdução

As redes ponto a ponto, ou redes peer to peer (também conhecidas como redes p2p) vêm se desenvolvendo rapidamente nos últimos tempos, popularizadas principalmente através de aplicações de file sharing (softwares de compartilhamento de arquivos em escala mundial), tais como o Napster, um dos precursores, e posteriormente o Edonkey, Kazaa, SoulSeek, Audio Galaxy, Gnutella, Freenet, BitTorrent entre outros. Apesar de atualmente se associar à tecnologia p2p a estas aplicações de file sharing, a tecnologia e as redes p2p não se originaram como necessidade delas.

Uma das primeiras necessidades de redes p2p se deu muito antes, muito provavelmente nas primeiras implementações de servidores DNS (Domain Name System) na década de 80, servidores que mapeiam endereços IP a domínios alfanuméricos. Tais servidores necessitavam e necessitam garantir que dos IPs mapeados para um domínio estão mapeados para apenas aquele domínio e que dois IPs de hosts distintos não mapeiam o mesmo domínio. Já como os servidores DNS trabalham em escala global, eles devem se comunicar para atingir estes objetivos. Mas se comunicar de que forma? Dai surgiu a necessidade de uma rede p2p bem estruturada

Assim como o DNS, o principal problema destas implementações de file sharing de redes p2p é que por serem pioneiras não utilizavam estudos sólidos na área. Muitas destas implementações e primeiras versões do DNS utilizavam (e ainda utilizam) uma rede p2p classificada como rede híbrida: um servidor central tem como base indexar os nós da rede e fazer buscas em rede ou buscas locais. A abordagem p2p neste tipo de rede estava apenas na comunicação entre nós na rede, enquanto o servidor tinha todo o trabalho de localizar e preparar conexões entre nós.

Um dos principais problemas desta implementação é bem visível: o grande gargalo gerado na rede pelo servidor. Outro problema grave é que assim como uma rede

topologia estrela, a queda de um nó central é crítico para o funcionamento da mesma, tornando-a muito suscetível a ataques (ataques DoS, por exemplo) e falhas.

Como soluções a estes problemas foram desenvolvidos estudos em redes que evitassem esta centralização, as chamadas redes descentralizadas ou redes peer to peer puras. Tais redes têm como base além da descentralização a idéia de que todo nó na rede tem mesma prioridade: nenhum nó tem funcionalidade diferente ou é mais importante que outro.

São várias as vantagens deste tipo de rede:

- Escalabilidade: Pela rede ser distribuída e descentralizada, evita-se o grande gargalo dos nós principais (servidores, nós chefe). Na maioria destas redes para cada nó só é necessário conhecer apenas $\log(N)$ dos nós da rede, com $N =$ número total de nós na rede.

- Balanceamento de Carga: O trabalho tende a ser dividido igualmente entre nós assim quanto maior a quantidade de nós na rede, maior a divisão de trabalho.

- Disponibilidade e Robusteza: A queda de um nó qualquer na rede não causa grande impacto, apenas um pequeno impacto local, tornando a rede muito mais resistente a ataques e falhas. Quanto maior a quantidade de nós na rede mais resistente ela fica.

- Anonimato: Permite anonimato e pode tornar muito mais difícil traçar ou registrar as ações de um dado nó na rede.

- Flexibilidade: Um ou mais nós não precisam se manter conectados sempre na rede. Qualquer nó pode entrar ou sair da rede a qualquer momento.

2. Objetivos

O objetivo geral deste trabalho é a construção de um middleware posicionado entre uma aplicação que necessita de dados e a fonte de dados. Também tem como objetivo o estudo e comparação das diferentes redes p2p e seus protocolos, sendo escolhido um deles para implementação.

Normalmente os middlewares de distribuição de dados têm uma arquitetura semelhante à cliente x servidor, onde a aplicação atua como um cliente e todos os outros hosts servem como servidores. Normalmente o conjunto de servidores físicos é transparente ao cliente, que enxerga apenas um servidor lógico.

Por exemplo, em um banco de dados distribuído, a aplicação o acessa via algum conector e realiza transações com ele. Atrás deste conector existe um DBMS (Database Management System) que além de monitorar uma série de entidades, se encarrega de definir regras de como e para quais hosts é encaminhada a transação gerada pelo cliente, realizando a transação de forma distribuída.

Neste cenário o conector e principalmente o DBMS agem como um middleware que deixa a fonte de dados transparente à aplicação. As databases distribuídas entre os diferentes hosts caracterizam a fonte de dados física. A fonte de dados que a aplicação enxerga caracteriza a fonte de dados lógica.

No middleware proposto neste trabalho, a situação descrita acima também pode ocorrer: uma fonte de dados lógica para uma aplicação (figura 1.1) é distribuída fisicamente entre diferentes hosts via rede TCP/IP (figura 1.2). O grande diferencial deste middleware é que, assim como a aplicação é um nó na rede que utiliza a fonte de

dados caracterizando um cliente, ele utiliza da descentralização proposta pelas redes p2p fazendo com que o host desta aplicação também possa ter uma fonte de dados própria acessada por outros clientes, caracterizando um servidor.

O que se pode obter no final é uma rede de dados globais e distribuídos onde cada host pode executar uma aplicação distinta com uma única fonte de dados lógica global, atuando tanto como servidor como cliente.

Outra diferença é que por ser um middleware voltado a Internet e que utiliza redes p2p descentralizadas, é esperado que a rede seja dinâmica, que sua topologia possa mudar com o tempo e que nós entrem e saiam da rede de forma não determinística, criando uma distribuição de dados de alta disponibilidade.

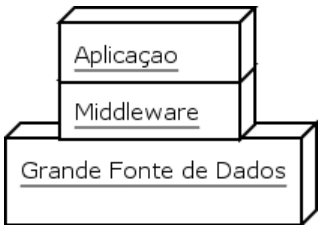

Figura 1.1 A aplicação enxerga a fonte de dados como única e local.

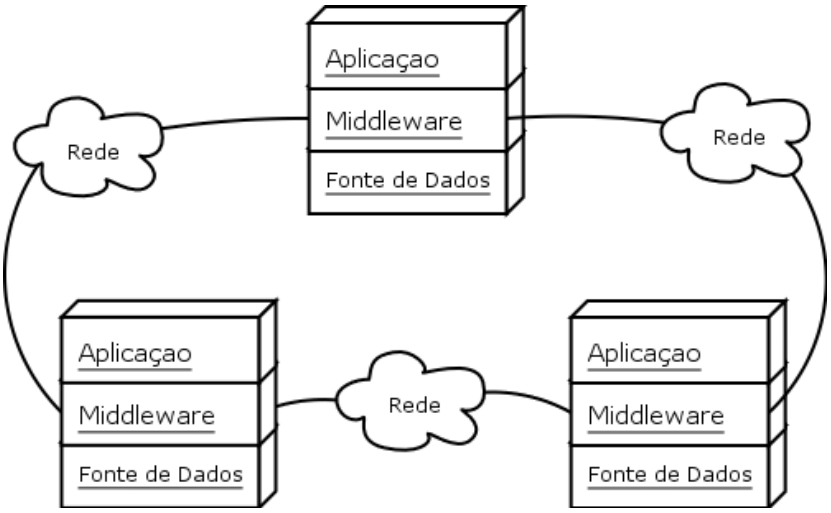

Figura 1.2 A visão física de como é estruturado o middleware, dividindo a fonte de dados em diferentes hosts que também executam suas próprias aplicações.

Exemplificações de uso deste middleware seriam: serviço de instant messaging ou e-mail, implementação de servidores tipo DNS, aplicações de file sharing ou qualquer aplicação que necessite de uma grande quantidade de dados, distribuição de carga, distribuição de dados, replicação de dados, compartilhamento de dados, comunicação direta, alta disponibilidade ou escalabilidade.

Este middleware pode lembrar o middleware JXTA[5], mas a grande diferença entre o JXTA e este middleware é que, além do tamanho e da abrangência, o JXTA implementa uma rede p2p híbrida enquanto este implementará uma rede p2p pura/descentralizada.

3. Peer to Peer

O middleware proposto irá essencialmente implementar uma rede p2p, mas tal rede deve seguir um protocolo. Tal protocolo, suas bases e suas características serão apresentados nesta seção.

3.1. Consistent Hashing e DHTs

Em uma tabela *hash* normal mapeada via uma função congruente do tipo $x \to (ax+b) \bmod p$, quando se muda o tamanho do espaço de *hashing* p, todos os elementos da tabela ganham um novo *hash*. Quando isto é aplicado sobre a Internet, onde o espaço de *hashing* p pode mudar constantemente com a saída e entrada de *hosts*, torna-se impraticável utilizar desta função *hash* normal para mapear estes *hosts*. Para toda mudança de topologia ou estado de rede, todos os *hosts* mapeados deveriam ser remapeados.

Consistent hashing[1] veio como forma de minimizar estes remapeamentos, tornando-os locais (dentro apenas de seus "*buckets*") invés de globais, assim possibilitando um *hashing* mais seguro mesmo em ambientes dinâmicos. DHTs são formas de distribuir uma tabela *hash* feita via *consistent hashing* entre vários *hosts*.

3.2. Kademlia

Apresentada em cerca de 2001, Kademlia[2] é definida como "*peer-to-peer distributed hash table* (p2p DHT)" ou então como tabela *hash* distribuída ponto a ponto. Ela garante boa consistência e performance mesmo em um ambiente com possíveis falhas. Ela utiliza uma métrica XOR para calcular distâncias que facilita o algoritmo e conseqüentemente facilita sua implementação.

Assim como no Chord[3], cada nó na rede recebe um identificador único de 160 bits que pode ser a criptografia SHA-1 de um conjunto de dados que identifica um *host*, como por exemplo o endereço IP e porta, ou então endereço MAC da placa de rede.

Os dados são salvos em duplas no formato <chave, valor>, sendo a chave também um *hash* de 160 bits do identificador do dado, como por exemplo, seu nome. Tais dados são salvos em nós que tem seu identificador "próximo" ao da chave, sendo esta noção de proximidade dada pela operação XOR bit a bit (ou exclusivo) entre o identificador do nó e a chave do dado.

Semelhante a topologia de servidores DNS, os nós na rede Kademlia são estruturados em forma de árvore binária, onde cada nó guarda seus nós vizinhos dentro de "*buckets*". Cada *bucket* representa uma sub-árvore de todo o sistema a qual o nó dono do bucket não participa. (figura 2). Esta estrutura garante que se cada nó possui pelo menos um nó conhecido em cada um de seus $\log(N)-1$ buckets, pode-se alcançar qualquer nó da rede. No caso do Kademlia, cada nó tem 160-1 buckets, pois cada nó tem 160 bits de identificação.

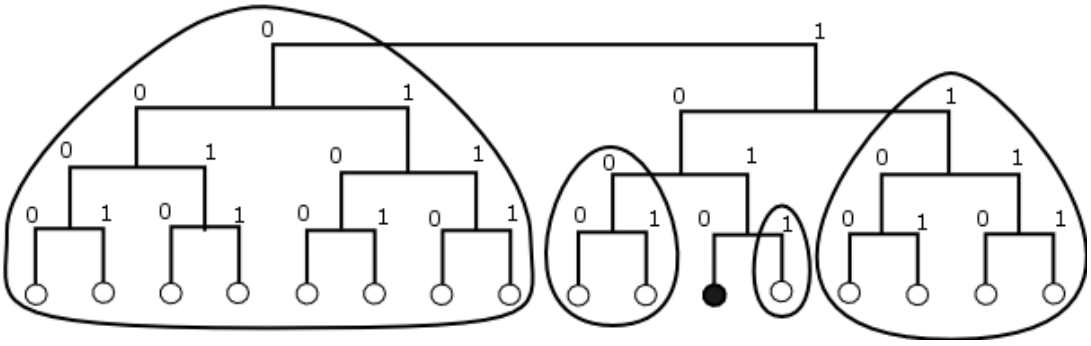

Figura 2 Exemplo de uma árvore Kademlia com identificadores de 4 bits. Cada folha representa um host. Os círculos são os buckets para o nó 1010 (nó negro), onde cada bucket é uma sub-árvore a qual o nó não pertence.

3.2.1. Protocolo

O protocolo Kademlia define 4 tipos de RPCs (*Remote Procedure Call*): PING, STORE, FIND_NODE e FIND_VALUE:

PING verifica se um dado nó está *online*.

STORE diz para um dado nó armazenar um par <chave, valor> para leitura posterior.

FIND_NODE recebe como argumento o identificador de um nó, realizando uma busca e retornando de forma assíncrona uma lista de nós próximos ao nó sendo buscado.

FIND_VALUE é semelhante ao FIND_NODE com única diferença que caso um nó que receba a RPC tenha o dado, o retorna.

Todas as mensagens contêm o nó identificador do remetente embutido, assim evitando mensagens de configuração, pois o nó destinatário pode utilizar qualquer mensagem que chegue a ele para conhecer a topologia da rede. Estas mensagens também possuem um identificador de RPC aleatório criado pelo remetente e que deve ser respondido pelo destinatário, para evitar forjamento de endereços.

Para fazer buscas no sistema (*lookups*) o remetente envia buscas via FIND_NODE de forma recursiva para nós no *bucket* no qual o destinatário pertence ou deveria pertencer até chegar no destinatário. Para que um nó possa buscar em qualquer nó dentro do sistema é necessário que cada *bucket* do nó tenha pelo menos 1 nó conhecido, assim sendo necessário conhecer apenas $\log_2(N)-1$ nós da rede para se ter comunicação total, com $N =$ quantidade total de nós.

Estas são as funcionalidades básicas da Kademlia, mas ela pode prover uma série de outras vantagens como *caching* de dados, replicação de dados, divisão de carga e certa resistência a forjamento de endereços e ataques DoS.

4. Implementação

A linguagem Java foi escolhida para implementação deste middleware, pois possui uma série de facilidades de desenvolvimento, faz uma abstração de uma série de problemas de baixo nível para o desenvolvedor, pois é voltada para o alto nível, é portável e tem ganhado cada vez mais espaço no mercado e no mundo acadêmico.

Após analise de protocolos levando em conta aspectos como escalabilidade, disponibilidade, robusteza e complexidade, três protocolos tornaram-se interessantes para implementação: Kademlia, Chord e Pastry[4]. Devido ao fato da Kademlia se apoiar em conquistas do Chord e do Pastry, ela conseguiu melhores ou iguais resultados e dispõe de uma quantidade maior de funcionalidades. Kademlia pode ser vista como uma evolução destes outros dois protocolos e por este motivo foi escolhida para implementação.

Este middleware comunica-se basicamente com três entidades: a aplicação que o utiliza, a fonte de dados própria do host e a rede de comunicação para acesso às outras fontes de dados.

A partir destas três entidades são providos dois tipos de serviços que podem ser encarados como cliente e servidor: atendimento das requisições da aplicação e atendimento das requisições de outros hosts da rede (sendo tratado como atendimento à rede).

No atendimento da aplicação é quando são geradas requisições à rede e à fonte de dados, atuando como cliente enquanto a rede e a fonte atuam como servidor. No atendimento à rede ocorre a situação inversa: o middleware atua como servidor da rede, utilizando a fonte de dados e provendo dados à rede sem o conhecimento da aplicação.

Em primeira instância o foco principal é na terceira entidade, a rede, pois é o que a Kademlia especifíca e é a essência do projeto. O acesso a dados e comunicação com a aplicação são decisões independentes da Kademlia e serão definidas neste trabalho em segunda instância.

4.1. Comunicação entre hosts

Para a comunicação entre diferentes *hosts* existe a opção de se utilizar o RMI[6] (*Remote Method Invocation*) provido pela API do Java, que possibilita a chamada de métodos entre *hosts* de forma bem facilitada. O problema de se utilizar o RMI nesta situação é que para toda chamada RMI espera-se um retorno único (relação 1:1 entre chamada e retorno) enquanto na arquitetura do Kademlia, uma chamada a outro *host* pode retornar mais de um conjunto de dados de forma assíncrona e não determinística (relação 1:N entre chamada e retorno) ou então simplesmente não retornar dado algum.

Para se utilizar o RMI neste cenário, deve-se forçar uma chamada RMI para cada retorno que um *host* pode prover, forçando uma relação 1:1 para cada um dos N retornos. Isto além de aumentar a complexidade do software, degrada sua qualidade devido a maior quantidade de mensagens multiplicadas pelo *overhead* gerado pelo RMI.

Pelo fato do RMI não ser a melhor escolha para este software, foi escolhido o par sockets e serialização de objetos (*object serialization*) onde estes objetos representam transações entre os *hosts*. Apesar desta alternativa ser mais baixo nível, onde se deve

definir um protocolo entre os *hosts*, controlar o fluxo de mensagens etc, esta opção deixa o desenvolvedor livre de muitas restrições e assim pode focar em uma implementação fiel a Kademlia.

Apesar da grande maioria das aplicações utilizarem conexões TCP (inclusive o RMI), pois para cada requisição espera-se uma resposta, Kademlia utiliza pequenas mensagens assíncronas em grandes quantidades para diferentes *hosts*, onde não se espera respostas de todos os *hosts*.

Nesta situação conexões TCP não caracterizam uma boa escolha, enquanto que mensagens UDP são mais adequadas: são mais leves gerando menor overhead na rede, não abrem conexão evitando que o software necessite gerenciá-la e consequentemente degradando a performance, a perda de um pacote não causa impacto no software (sendo ate uma situação esperada em certas ocasiões) e nem sempre se espera resposta de uma mensagem enviada.

Na prática a grande maioria das redes *peer to peer* utilizam UDP devido ao pequeno tamanho das mensagens e grande quantidade de trocas de mensagens diretamente entre *hosts*.

4.2. Gerenciamento

Para lidar com as requisições vindas da rede será utilizada uma espécie de receptor que escuta a porta UDP esperando pela chegada de um pacote. Ao receber um pacote este receptor o transforma em um objeto do tipo transação e o enfileira em um buffer, atuando como um produtor.

Do outro lado do buffer existe o consumidor, que é um gerenciador que identifica se a transação é uma resposta de uma requisição anterior ou se é uma requisição vinda da rede. Caso seja uma resposta a uma requisição anterior, este gerenciador se encarrega de delegar a transação à *thread* que criou a requisição através de uma notificação. Caso a transação seja uma nova requisição vinda da rede, este gerenciador identifica qual o tipo de *RPC* que esta transação representa e delega para uma nova *thread* realizar o trabalho.

Nesta implementação percebe-se que para cada *RPC* é utilizada uma *thread* e como elemento de otimização pode-se criar *pools* em alguns pontos: pool de transações trabalhando em conjunto com o receptor, caso estas transações comecem a elevar seu custo de criação ou carreguem uma quantidade considerável de dados e um pool de *threads* trabalhando em conjunto com o gerenciador para evitar a criação de uma nova *thread* para cada requisição vinda da rede.

Cada *thread* criada pelo gerenciador deve gerenciar o status da RPC a qual ela representa recebendo eventos do gerenciador e podendo criar outras *RPCs* conforme necessidade. Ao criar uma RPC, a *thread* solicita o envio de uma transação a um emissor, que se encarrega de enfileirar as transações e enviá-las via porta UDP, este podendo ser um componente *multi-threaded.*

4.3. Fonte de Dados

Para se construir uma estrutura amigável a da Kademlia, a interface com a fonte de dados de cada *host* é em forma de uma tabela *hash* normal com tamanho predefinido, utilizando do par <chave, valor>, onde valor é resultado de uma operação de escrita nesta fonte.

Por trás desta interface pode ser utilizado qualquer tipo de fonte de dados, como um banco de dados, arquivos do tipo CSV (*Comma-Separated Values*), *properties*, *config* ou qualquer outro formato, uma região de memória ou ate então uma nova camada distribuída de dados.

4.4. Dificuldades e possíveis soluções

Uma dificuldade em nível de rede é que pelo fato do middleware utilizar UDP para sua comunicação, os *NATs* (*Network Address Translation*, também conhecido como *Network Masquerading*), largamente utilizados para mapeamento de uma ou mais redes internas em um IP em uma rede externa, podem se tornar um problema, pois são voltados em sua maioria a conexões TCP e têm de abrir exceções a pacotes UDP.

Uma das soluções a este problema é a utilização de um cliente STUN (*Simple Traversal of UDP over NATs*) que se conecta a um servidor STUN para descobrir quais os NATs presentes entre uma conexão entre dois pontos. O problema disto é que este servidor STUN não deve ser centralizado, pois isto vai contra o propósito do middleware.

Esta centralização e utilização do servidor STUN não pode ser eliminada, porém pode ser minimizada com a utilização de *caches* e a distribuição em servidores STUN em *hosts* da rede que não estão atrás de NATs.

Uma dificuldade em nível de implementação é o mundo não convencional imposto pela Kademlia. Isto aumenta a complexidade de implementação, pois as RPCs são executadas em paralelo em uma estrutura recursiva e independente: muitas RPCs podem ser executadas ao mesmo tempo, sendo que cada RPC pode criar novas RPCs de forma assíncrona enquanto continua sua execução, tendo que ao mesmo tempo monitorar todos os status de novas RPCS e RPCs anteriores, caracterizando um ambiente massivamente paralelo e voltado a eventos.

Em uma estrutura recursiva comum e síncrona temos que cada chamada recursiva é empilhada e entra em espera ao se criar uma nova chamada. Cada chamada criada é empilhada ate um ponto que se obtenha o retorno de uma chamada, assim esvaziando a pilha com este retorno sendo passado para cada chamada até chegar à primeira delas.

No mundo recursivo imposto pela Kademlia, cada chamada recursiva não caracteriza um empilhamento de chamadas, mas a paralelização de processos. Já como cada chamada espera por mais de um retorno assíncrono, ela não para seu processamento ao criar uma nova chamada. Para cada retorno obtido ela ainda pode criar uma nova chamada assíncrona, caracterizando uma árvore de chamadas onde nós pais tanto quanto nós filhos são executados em paralelo com a opção de criar mais processos. (figura 3)

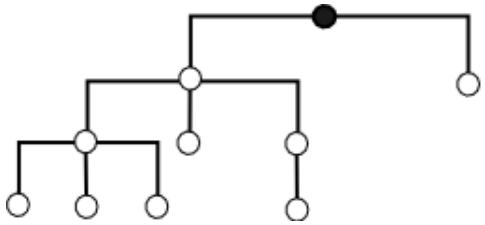

Figura 3 Exemplo de execução: Cada nó representa uma thread. O nó negro atua como o gerenciador que dispara novas threads ou encaminha a requisição para a thread encarregada. Cada thread pode disparar novas threads ou finalizar sua execução a qualquer instante. Quando uma thread finaliza sua execução ela finaliza suas threads filhas em cascata.

4.5. Validação

Por este middleware implementar a Kademlia, é esperado que ele possibilite para a aplicação as mesma facilidades que a Kademlia possibilita. Também é esperado que este middleware implemente as funções da Kademlia corretamente e que eventuais problemas que exijam modificações em nível de implementação não alterem a funcionalidade envolvida ao problema. Também é esperado que o middleware tenha uma performance aceitável e que seja previsível.

Para efetuar todas estas validações serão desenvolvidos basicamente três tipos de testes: testes unitários, testes de integração e testes de performance.

Testes unitários têm como objetivo validar uma dada funcionalidade, sem influência de agentes externos ou outros agentes internos. Exemplo: um teste unitário de busca por um nó na rede deve apenas testar a busca e mais nada. A construção do identificador do nó, a topologia da rede, o estado atual dos buckets, o estado dos nós e qualquer outro módulo ou configuração envolvida devem ser preparados de forma que estejam corretos formando um ambiente que não interferirá no teste.

Testes de integração têm como objetivo validar a integração entre funcionalidades (previamente testadas via teste unitário) ou então a integração do middleware com outras camadas como a camada da aplicação ou da fonte de dados.

Testes de performance têm como objetivo descobrir limites do software podendo ser de dois tipos: teste de carga, onde é verificada a resposta do software a uma grande quantidade de trabalho num curto espaço de tempo e teste de stress, onde o software é submetido a uma grande quantidade de trabalho porem distribuído em um grande espaço de tempo, muitas vezes simulando um acesso real ao software e vendo por quanto tempo o software continua em execução.

Pelo fato destes testes serem executados muitas vezes da mesma forma na produção de um software, automatizá-los pode ser muito útil. Para esta automatização pode ser utilizado o JUnit[7] para testes unitários e de integração e para testes de performance pode ser utilizado o JMeter[8], ferramenta destinada a stress de aplicações web e que pode ser adaptada para o stress deste middleware.

5. Conclusão

A produção deste middleware envolve um bom conhecimento em redes ponto a ponto e nos protocolos desenvolvidos em DHTs, especialmente o Kademlia, escolhida pela sua simplicidade comparada a outros protocolos e por oferecer melhor qualidade e maior quantidade de recursos. Boa parte deste conhecimento foi adquirido na fase inicial do projeto, fase caracterizada pela pesquisa na área.

Em seguida utilizando conhecimentos prévios em programação e engenharia de software, foi definida uma arquitetura em Java para a construção deste middleware.

As próximas etapas serão dedicadas à implementação e formalização desta arquitetura. Em paralelo a esta implementação serão construídos testes para validar os pequenos módulos do software e o software integrado.

Por último, mas não menos importante, a principal motivação para a construção deste middleware é o potencial visto nas atuais redes e aplicações de file-sharing e a tendência atual pela descentralização e processamento paralelo para se adquirir principalmente alta performance, disponibilidade e escalabilidade.

6. Referências

[1] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin e Rina Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web*

[2] Petar Maymounkov e David Mazières. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. http://www.cs.rice.edu/Conferences/IPTPS02/109.pdf

[3] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek e Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications. http://pdos.csail.mit.edu/chord/

[4] Pastry: A scalable, decentralized, self-organizing and fault-tolerant substrate for peer-to-peer applications. http://research.microsoft.com/~antr/Pastry/

[5] JXTA: Java platform-independent protocol for peer-to-peer networking. Sun Microsystems. http://www.jxta.org/

[6] Java Remote Method Invocation (Java RMI): http://java.sun.com/products/jdk/rmi/

[7] JUnit: Testing Resources for Extreme Programming. http://www.junit.org/

[8] Apache JMeter: a 100% pure Java desktop application designed to load test functional behavior and measure performance. http://jakarta.apache.org/jmeter/