UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

RENAN SARCINELLI
RODOLFO VIEIRA VALENTIM

PROGRAMAÇÃO EM SOCKETS PARA REDES P2P

VITÓRIA
2016

Introdução

Este trabalho consiste em uma implementação didática da rede proposta pelo artigo “Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications”[1] em uma forma mais simplificada.

Uma rede *peer to peer* consiste de uma rede que cada participante da rede, *peer*, tem papéis e responsabilidades iguais na rede, sendo eles responsáveis por manter e gerenciar a rede, sendo assim os arquivos compartilhados na rede ficam distribuídos entre os *peers*. Ao contrário da arquitetura cliente-servidor onde um dos membros da rede, o servidor é responsável por manter e gerenciar a rede, e os arquivos ficam centralizados nele. Devido a esta característica as redes *peer to peer* ficaram muito populares para compartilhamento de dados como arquivos de áudio, vídeo, multimídia ,entre outros.[2]

Nesta versão simplificada as funcionalidades JOIN, LEAVE, LOOK UP e UPDATE devem ser implementadas na rede. Cada uma delas será descrita durante a explicação das funcionalidades do programa.

Resumo

Redes P2P são redes onde todos os nós possuem a mesma importância, bem como as mesmas funções. Essas características levam a diversas vantagens como: armazenamento redundante, estabilidade, seleção de pares mais próximos, anonimidade, procura e autenticação. Chord é um protocolo para redes P2P que aceita apenas uma operação: mapeia uma chave para um nó e dependendo de aplicação um nó pode armazenar uma chave relacionada a algum conteúdo.

Existem diversas abordagens para o mapeamento que possuem vantagens e desvantagens em relação ao Chord: DNS, Freenet, Ohaha, Globe, CAN e outros. A ideia de Chord é de simplificar os sistemas P2P. O protocolo foca nas seguintes características: balanceamento de carga, descentralização, escalabilidade, disponibilidade e designação flexível.

De fato, o protocolo Chord específica como localizar chaves, como nós se conectam a rede e como se recuperar de falhas na rede. Para mapear uma chave para um nó, Chord usa “consistent hashing” que é responsável pelo balanceamento de carga. Consistent hashing, pode ser por exemplo SHA-1 (Simple Hashing Algorithm) que produz sempre uma chave de 160-bit.

Para a localização de uma chave na rede, basta cada par armazenar a dados sobre o sucessor, entretanto, cada procura teria de, na média, passar por O(N/2), por isso cada nó possui informações adicionais que são armazenadas na forma de tabela (finger table) com *m* linhas. A construção da tabela se dá da seguinte forma:

$$finger[k] = (n + 2^{k-1}) mod\ 2^m \quad 0 \leq k \leq m$$

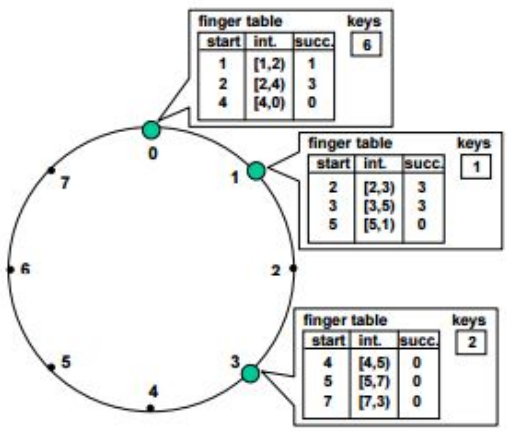

Figura 1 - Rede implementada vs implementação Chord

Para que ocorra tudo certo durante a operação, durante a inserção deve-se seguir alguns

passos.

- inicializar antecessor e sucessor;
- atualizar a finger table de todos os nós com a adição.

Para isso é necessário que seja informado algum endereço de nó na rede por algum agente externo. A inicialização do antecessor e sucessor é feita pela procura (look up) da posição onde determinada chave entraria na rede. Existe uma otimização do processo onde durante a procura, o nó mais novo copia toda sua finger table e tenta induzir a sua própria. Em seguida, o nó deve atualizar todas os nós que deveriam apontar para ele. Agora que a estrutura de rede está completa com a adição do nó, deve-se mover-se os dados dos quais ele é responsável.

Uma preocupação prática bem importante é com relação a concorrência e como fato de nós saírem voluntariamente ou não. Estabilização é uma técnica que permite a adição de nós manter o acesso a todos os nós, entretanto há a limitação de não conseguir corrigir redes que se bipartem ou uma rede que dá múltiplas voltas no espaço identificador. A estabilização ocorre de tal forma que, periodicamente, manda uma mensagem para seu sucessor perguntando pelo seu antecessor e atualizando até que a resposta seja ele mesmo.

Para se recuperar de falhas, uma alternativa é onde cada nó armazena uma lista de sucessores. Esta lista de sucessores também pode facilitar camadas mais altas na replicação de dados.

A fim de avaliar características importantes do protocolo Chord alguns experimentos e simulações foram realizados. Os principais aspectos avaliados da rede foram:

- Balanceamento de carga. Indica se as chaves estão bem distribuídas entre os nós, se há nos muito sobrecarregados ou não. Estes testes demonstraram o comportamento desejado da rede distribuindo a carga, com poucos nós sobrecarregados.
- Tamanho do caminho. Distância entre dois nós na rede, neste caso específico foi medido o número de lookups até encontrar o nó desejado. Com os resultados deste teste foi possível concluir que o tamanho do caminho cresce Logaritmicamente em relação ao crescimento linear do número de nós
- Falhas simultâneas em nós. Mede a capacidade de a rede se recuperar após a falha de diversos nos simultâneos. Para este caso específico foram medidos o número de falhas em lookups após a estabilização devido a falha de nós simultâneos. Após análise das simulações foi constatado que o número de falhas nos lookups percentualmente era quase que diretamente a porcentagem de nós que haviam falhado.
- Lookups durante estabilização. Este teste avalia o comportamento do protocolo em situações de falhas, joins e leaves frequentes, interferindo fortemente no processo de estabilização da rede. Para isso foram medidos novamente as falhas de lookup em dadas circunstâncias

- Resultados experimentais. Para validar a implementação da rede na internet foi realizado um teste do protocolo na rede de internet real. Neste teste foi medida a latência de um lookup de acordo com o número de nós da rede.

De acordo com os autores, trabalhos futuros para a melhora do protocolo podem atacar o fato de a rede não consegue se recuperar de falhas que particionar a rede. Outro ponto a melhorar é quanto a vulnerabilidade a ataques onde nós mais intencionados podem afetar na disponibilidade de dos dados. Pode-se, também, haver esforços na tentativa de diminuir a latência do look up.

Conclui-se que Chord é uma abordagem que soluciona o problema que muitas redes P2P têm na hora de encontrar um dado na rede e Chord o resolve de uma forma descentralizada, simples, escalável e robusta.

Descrição de Implementação

Para a implementação da aplicação de rede, escolheu-se a linguagem de programação Python (versão 2.7.12) por ser de alto nível e focar em legibilidade e ser extremamente portável, sendo possível executar seus *script* nos principais sistemas operacionais (Windows, Linux e MacOS) e até nos sistemas para dispositivos mobile com (Android e iOS). Favorecendo-se da flexibilidade de paradigmas do Python, implementou-se a aplicação em utilizando tanto a orientação a objetos quanto o funcional.

A aplicação se baseia na troca na mensagens entre os nós de rede usando o protocolo UDP. Python, através da biblioteca socket [3], fornece métodos para mandar um pacote definido pelo usuário para um determinado IP, assim como receber de um socket associado a uma porta, no caso 12233. Como a troca de mensagens é constante, criou dois métodos que encapsulam todas as etapas e facilitam a programação, evitando retrabalho.

Recebimento e envio de pacotes

O método “sender” envia um pacote para um determinado contato. O método é sempre chamado por algum método de requisição. Os métodos de requisição são bloqueantes para evitar problemas de concorrência.

```
func sender (contato, dados)
   socket = criar(IPv4, User Datagram) # cria um socket para UDP
   socket.enviar_para(dados, contato)
   socket.fechar()
```

Pseudocódigo 1 - Função que envia os pacotes.

```
func metodo_requisição (contato, dados)
   fmtdados = formatar(dados)
   sender(contato, fmtdados)
   enquanto(dados_recebidos = listener.resposta() == NULL)
      continue
   retorno dados_recebidor
```

Pseudocódigo 2 - Método de requisição na rede. A função formatar foi usada por simplicidade. Na implementação é necessário que cada mensagem seja formatada como especificada no protocolo. Percebe-se que o bloco “enquanto” bloqueia o processamento enquanto não receber resposta.

O método “listener” tem um grau de complexidade a mais comparando com o

“sender”, pois é esperado que um nó esteja sempre disponível para receber uma mensagem, portanto foi necessário o uso de thread uma vez que o programa também precisa estar disponível para receber solicitações do usuário. Deste requerimento surge um complicador, caso o usuário faça uma requisição na rede e ao mesmo tempo recebe-se um requisição, em uma abordagem inocente, trataria a requisição como a resposta de sua pergunta. Assim, definiu-se que o “listener” estaria sempre escutando a rede na porta definida e qualquer mensagem recebida será encaminhada para seu respectivo tratador. Sempre que é esperada alguma resposta, o nó informa ao “listener”de pergunta e uma vez recebida a resposta, esta é encaminhada ao seu tratador e informa ao nó do ocorrido.

```
func listener (ip, porta)
   socket = criar(IPv4, User Datagram)     # cria um socket para UDP
   socket.associa(ip, porta_padrão)        # Associa o receptor a uma porta e ip

                                           # como o listener roda em uma thread separada
   enquanto (vida == verdadeiro)           # cotrolar quando deve-se sair de um loop
       contato, pacote = socket.recebe_de()# tentar receber de porta padrão
       se (pacote == resposta_requisicao)
           metodo_requisicao(contato, pacote)
       se (pacote == requisicao)
           metodo_resposta(contato, pacote)
```

Pseudocódigo 3 - Função de recebimento de pacotes na rede. Roda em uma thread diferente do programa principal, assim o loop fica para sempre verificando a chegada de mensagens. Sempre que a thread principal quiser encerrar sua atividade, basta mudar a informar por meio de variável ‘vida’.

```
func metodo_resposta (contato, dados)
   dados = desformatar(dados)
   contato, fmtdados = processar (dados)
   sender(contato, fmtdados)
   retorno dados_recebidor
```

Pseudocódigo 4 - Método que processa as requisições da rede ao nó. Sempre que receber um requisição os dados são decodificados, processados de acordo com as regras e encaminhados ou ao próximo ou ao contato que fez a requisição. Percebe-se que não é uma função que não bloqueia a aplicação.

Pesquisar posição de inserção na rede

Umas das funções principais na rede é a de LOOK UP que consiste em uma busca linear (de nó a nó) procurando pelo nó que tem a chave de identificação imediatamente após a chave procurada. Vale uma observação importante, o protocolo não informa campo para mensagem de erro, portanto, sempre deve-se retornar uma resposta válida. Caso se procure

uma chave que já existe na rede, a aplicação de rede deve retornar o sucessor do nó de mesma chave.

Para se fazer um LOOKUP, a aplicação usa o ”método_requisição” especificado no Pseudocódigo 2. O processamento de mensagem quando recebida uma requisição de LOOK UP é tal qual ilustrada no Pseudocódigo 5 e é esta que é chamada .

```
func processar_pedido_lookup(chave_procurada, origem_da_procura)
    se (chave_procuradat >= minha_chave
        e (chave_procurada < chave_sucessor ou chave_sucessor <= minha_chave))
        ou (chave sucessor <= minha_chave e chave_procurada < chave_sucessor)
          sender(origem_da_procura, ip_sucessor)
    senão
          sender(ip_sucessor, chave_procurada)
```

Pseudocódigo 5 - Há muitas considerações feitas para saber se a chave procurada, que levam em conta o fato de ser uma lista circular.

Atualizar informação na rede

Também é necessária a atualização na rede. Está mensagem é de certa forma mais simples. A mensagem é enviada pelo novo sucessor ao antecessor para avisá-lo. O recebimento e envio de mensagem usá os Pseudocódigos 2 e 4.

Conectar-se a uma rede

Para conectar-se a uma rede é preciso primeiro se ter um contato na rede. Envia-se uma mensagem de LOOK UP de uma chave de 32 bits gerada aleatoriamente. Obtida a resposta, envia-se uma mensagem de JOIN para o sucessor e em seguida uma mensagem de UPDATE para o novo antecessor. Caso alguma destes métodos retorne erro por inconsistência, se é gerada uma nova chave e reinicia-se o processo.

```
func conectar(chave, contato_ip):
   faça:
        ip_sucessor = lookup(chave, (contato_ip))
        enquanto (join(ip_sucessor) e update(novo_antecessor))
```

Pseudocódigo 6 - O ‘enquanto’ processa primeiro o join e em caso de erro recomeça o loop, em caso de sucesso, faz-se o update e se ocorrer tudo como espera, sairá do loop.

Desconectar de uma rede

Para desconectar-se de rede, basta enviar mensagens de “leave” para o sucessor a antecessor. Ao receber esta mensagem, o nó deve responder uma mensagem de confirmação. Importante notar que o método é bloqueante, uma vez que sair é uma função

crítica e não faz sentido manter qualquer outro processamento enquanto se é executada tal função.

```
func desconectar():
   sender(ip_sucessor, fmtdados)
   enquanto(dados_recebidos = listener.resposta() == NULL)
      continue
   sender(ip_antecessor, fmtdados)
   enquanto(dados_recebidos = listener.resposta() == NULL)
      continue
```

Pseudocódigo 6 - O 'enquanto' processa primeiro o join e em caso de erro recomeça o loop, em caso de sucesso, faz-se o update e se ocorrer tudo como espera, sairá do loop.

Descrição dos Testes

Para validar a implementação tanto do programa quanto das aplicações da rede foram adicionadas as funções:

- INFO, que mostra as informações do nó e;
- LIST que mostra todos os nós da rede.

Além das mensagens dos pacotes recebidos e enviados assim como checkpoints importantes para execução adequada do programa. Foram testadas primeiro as funcionalidades da rede separadamente através do seguinte procedimento. Foram usados até 4 computadores rodando a aplicação:

- Um nó criou uma rede usando o CREATE;
- Checou-se com INFO se as informações são válidas;
- Um novo nó entrou na rede;
- Checou-se se os dados então corretos;
- O nó criador saiu de rede;
- Checou-se se a informações estão corretas e finaliza o teste.

Garantindo o sucesso das aplicações que deveriam ser implementadas pela rede a rede é iniciada para a próxima etapa de testes. Para testar o funcionamento de vários nós juntos foi adotado o seguinte procedimento:

- Primeiro um nó cria a rede usando a funcionalidade CREATE;
- Comando INFO foi executado verificando se os atributos do nós tinha sido devidamente atualizados;
- Em seguida, um nó se conecta a rede pelo nó criador e o próximo se conecta pelo nó mais novo. Assim até que todos os nós estejam conectados;
- Assim que todos estão conectados, utiliza-se o comando LIST que lista todos os elementos na rede. Em seguida, nós aleatórios vão deixando a rede e a cada LEAVE, executa-se o LIST e checando a consistência.

Testes para interoperabilidade foram feitos como descritos anteriormente. O primeiro teste funcionou como esperado, mas o segundo teste não foi possível devido a uma ambiguidade no protocolo. Uma vez que não se sabe o que deve-se retornar caso pesquise-se por uma chave que já existe na rede.

Conclusão

Concluiu-se que esta rede funciona muito bem como prova de conceito para implementação e execução de uma rede P2P. Entretanto, percebe-se que esta rede carece de tratamento de erros, como um nó sair de rede sem avisar ao seus pares, além de não ser tão eficiente do ponto de vista das operações, um nó apenas conhece seu sucessor e antecessor e encontrar um nó na lista custa em média O(N/2), sendo pouco escalável. A implementação descrita pelo protocolo Chord, por exemplo, segue a mesma ideia e melhora nos pontos citados anteriormente.

Referências

[1] CHORD: A Scalable Peer-to-peer Lookup Service for Internet Applications. San Diego, California, USA: SIGCOMM'01, 2001. Disponível em: <https://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf>. Acesso em: 18 set. 2016.

[2] HERLEY, Cormac. **Peer to peer network**. US 7343418 B2. 03 de jun. de 2002. [S.l.], p. 1 Disponível em: <https://www.google.com/patents/US7343418>. Acesso em: 18 set. 2016.

PYTHON SOFTWARE FOUNDATION. . Socket : Low-level networking interface. Disponível em:<https://docs.python.org/2/library/socket.html>. Acesso em: 17 set. 2016.