

Disciplina: Redes de Computadores

Trabalho: Implementação de protocolo P2P utilizando DHT e Sockets

Alunos:

Raul Vieira Cioldin – 379468

Gabriel Lucius Pomin – 379280

Samuel Carlos Machado Rodrigues – 296252

Edgar Fardin – 321680

Professor:

Luis Carlos Trevelin

Implementação

Linguagem: Python 2.7

Sistema Operacional: Arch Linux (Kernel 3.11)

Principais componentes (de linguagem):

- Sockets
- Pickle
- Threads

Arquivos:

- node_indexer.py
- supernode.py
- client.py
- hash_ring.py

Idéia geral:

Cada arquivo representa uma perspectiva da aplicação, exceto o “hash_ring”, que denota uma estrutura.

O “node_indexer” é um indexador de supernós à semelhança de um tracker (bittorrent) e ainda mais próximo em conceito ao servidor do protocolo implementado pelo extinto napster. Sua responsabilidade básica é servir aos supernós (e a eles apenas) informações de localidade na rede.

O “supernode”, que representa os supernós, exerce o papel (resumido) de repositório de arquivos, atendendo a requisições de clientes.

O “client” é a interface do cliente com a rede overlay, mais especificamente com um de seus supernós (e indiretamente com todos eles, por via de eventuais redirecionamentos).

A estrutura codificada em “hash_ring”, como o nome infere, é uma tabela hash em forma de anel que é utilizada no indexador (“node_indexer”) para registrar os nós e distribuir chaves entre eles, o algoritmo (descrito no Kurose) garante o chamado “consistent hashing”, grosseiramente falando, faz um balanceamento de chaves entre os nós considerado eficiente (ou seja, bem distribuído).

Explicação (detalhada) da solução:

Como especificado, a comunicação entre as partes da aplicação se dá via API de sockets, como optamos por TCP, primeiro é estabelecida a conexão entre dois pontos e, depois, uma das partes (ou ambas) atendem a requisições. Essas requisições foram convencionadas da seguinte forma:

- Requisitante envia um buffer (texto puro) onde o primeiro caracter é sempre uma operação (exemplo: 'q' + 'arquivo.txt', ou seja, 'qarquivo.txt')
- Quando convém, o servidor desse contexto envia um tipo de ACK e atende a demanda

Para estruturação da rede overlay, os supernós têm de adotar dois canais de comunicação (para melhor segregar essas funções, o supernó utiliza duas threads e duas portas):

- Controle
- Escuta

O canal de controle é utilizado pelo supernó para interagir com o indexador, a caráter permanente (ou até que um dos dois se desligue da rede, evidentemente).

O canal de escuta serve solicitações (únicas) de clientes.

Um terceiro canal foi construído a partir desses dois, de maneira transparente à aplicação, chamado de Sincronizador (também uma terceira thread). Sua responsabilidade é periodicamente varrer o repositório de um supernó, extraindo do canal de controle a necessidade ou não de se distribuir algum arquivo pela rede e, quando assim acontece, atua como cliente e solicita a um outro supernó o upload de arquivo (ou seja, utilizando o canal de escuta desse outro supernó).

No ato de conexão com o indexador, o supernó ainda não é registrado como um repositório, isso porque o indexador precisa manter em sua tabela hash a porta que será usada no canal de escuta do supernó, então ele aguarda a mensagem de “append” do supernó, no formato: 'a' + 'porta' (ex: 'a15500'), só então o supernó é cadastrado na tabela hash (uma string hostname:porta correspondente) e está passível de ser encontrado pelo mecanismo de indexação.

Outras operações interpretadas pelo indexador:

- “remove”, 'r' + 'porta', elimina o supernó correspondente da tabela hash mas não termina a conexão, porque é necessário que o supernó faça uma última sincronização e o serviço de indexação ainda é necessário para ter acesso à nova disposição das chaves da hash. Exemplo: 'r15500'.
- “disconnect”, 'd', antes de fechar o socket do lado do supernó, ele avisa o indexador de forma que o mesmo não tente mais escutar no socket correspondente (o que poderia deixar o indexador bloqueado esperando pelo timeout).
- “query”, 'q' + 'arquivo', a operação de consulta do indexador, dado um nome de arquivo, ele gera a chave hash (algoritmo md5) e adquire o supernó responsável pela chave, retornando ao supernó requisitante o 'host:porta' correspondente (o hostname e a porta de escuta que foi registrada no “append”). Exemplo: 'qArquivo.txt'.

Da mesma forma que o indexador, os supernós atendem a clientes por via de operações indicadas pelo primeiro caracter das mensagens (direcionadas a eles):

- “list”, 'l', o supernó atende a essa requisição transferindo sua lista de arquivos, essa operação em particular envolve a serialização de objetos da linguagem (ao enviar texto puro ou sequência de bytes nenhum tratamento em especial é necessário), o componente cPickle (a contraparte mais rápida do pickle) é usada para fazer um “dump” para bytes de um objeto python list, podendo assim ser transferido normalmente no socket (via diretiva sendall).
-

- “upload”, 'u' + 'arquivo', requisição de envio de arquivo para o supernó, um cliente ou a thread de sincronização de outro supernó deseja enviar um arquivo, a operação envolve um ACK de aceitação ou não, uma vez que o arquivo pode já existir, então, o solicitante aguarda 2 bytes de resposta do supernó que pode ser “OK” ou “NO”, onde ele informa respectivamente que pode ou não pode receber o arquivo. Exemplo: 'uArquivo.txt'.
- “download”, 'd' + 'arquivo', requisição de download de um arquivo da rede feito por cliente a um supernó, também envolve um ACK que pode ser de 3 instâncias: “OK”, “RE”, “NO”.
 - “OK” implica que o supernó contém o arquivo e vai enviá-lo.
 - “RE” informa ao cliente que o arquivo não está presente no supernó mas pode estar contido em outro, responsável pela chave gerada a partir do arquivo (informação extraída via consulta ao indexador como previamente explicado), enviando em sequência mensagem com o 'hostname:senha' do outro supernó para que o cliente trate do redirecionamento.
 - “NO” ocorre quando o arquivo não é encontrado na rede, o nó responsável pela chave foi alcançado e não contém o arquivo solicitado (lembrando que não poderia estar em outro nó de qualquer forma, graças à thread de sincronização).

Na perspectiva do cliente a aplicação é um serviço de repositório de arquivos, a rede overlay é transparente até certo ponto, para ser mais claro, um único supernó é acessível a tempo de execução exceto quando ocorre um redirecionamento na operação de download. O cliente, mediante conexão a um supernó (em seu canal de escuta), pode requisitar listagem, download ou upload como acima explanado.

Acerca da aplicação e da execução de seus módulos (programas), temos os possíveis argumentos:

- “node_indexer.py”
 - porta utilizada (padrão: 6110)
- “supernode.py”
 - caminho absoluto do repositório de arquivos (padrão: '/home/raul/estudos/redes/p2p_filesharing/repositorio01/')
 - hostname e porta do indexador (padrão: 'localhost' e 6110)
 - hostname, porta do indexador e caminho absoluto do repositório de arquivos
- “client.py”
 - hostname e porta do supernó (posicionais e obrigatórios)
 - operação (obrigatório, mutuamente exclusivo em suas opções)
 - '-l' ou '--list'
 - '-d' ou '--download', necessário explicitar o nome do arquivo
 - '-u' ou '--upload', necessário explicitar o caminho completo ou nome do arquivo (se o arquivo desejado estiver na raiz de execução do cliente)

Outras considerações:

A idéia básica de operação dos supernós (uso de dois canais, ACK's) foi extraída da RFC 959 (FTP) e o uso do anel hash para implementar a distribuição equilibrada de chaves estava no Kurose, o algoritmo original (não modificado) foi encontrado aqui: <http://amix.dk/blog/viewEntry/19367>.

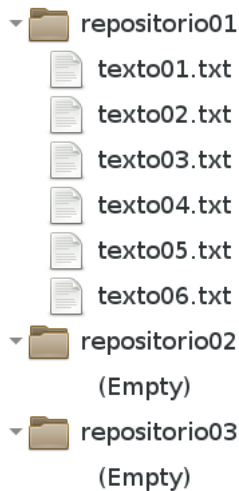
A comunicação TCP via socket em python é bem parecida com a sua contraparte em C mas para destacar uma vantagem, o uso do select em C é de uma complexidade muito maior (embora sabendo utilizar o select em Python não existe muita dificuldade em fazer o mesmo em C). A serialização de

objetos também foi muito facilitada com os métodos da API padrão do Python para construir e deconstruir em bytes. Uma outra forma de fazer o mesmo, uma vez que o único objeto que estou serializando é uma lista de strings, seria concatenar as strings com um delimitador e enviar no socket mas, se existisse um caracter delimitador na string que se quer enviar, outra camada de validação deveria ser codificada.

Vale ressaltar que só usei a aplicação no linux (e nenhuma biblioteca adicional do python 2.7). No LIG do dc algumas máquinas tem ubuntu (user: aluno, senha: Aluno2013).

Demonstração:

Primeiro, observe a estrutura de diretórios abaixo:



Cada uma dessas pastas serão usadas por um supernó, repare que apenas o primeiro repositório está populado com arquivos.

O que eu desejo mostrar com isso é que durante a sincronização vai ocorrer um balanceamento e os arquivos vão ser copiados do primeiro supernó para os demais.

Iniciando o indexador

node indexer.py:

```
[raul@raul-laptop p2p_filesharing]$ python2 node_indexer.py
Iniciando o servico na porta 6110
```

Iniciando 3 supernós

supernode.py [1]:

```
[raul@raul-laptop p2p_filesharing]$ python2 supernode.py localhost 6110 /home/raul/estudos/redes/p2p_filesharing/repositorio01/
Recebendo conexões na porta 33515
Superno conectado ao indexador (localhost, 6110) pela porta 37975
```

supernode.py [2]:

```
[raul@raul-laptop p2p_filesharing]$ python2 supernode.py localhost 6110 /home/raul/estudos/redes/p2p_filesharing/repositorio02/
Recebendo conexões na porta 42075
Superno conectado ao indexador (localhost, 6110) pela porta 37977
```

supernode.py [1]:

```
[Balanceamento] Retransmitindo texto01.txt a (127.0.0.1:42075)
[Balanceamento] Retransmitindo texto02.txt a (127.0.0.1:42075)
[Balanceamento] Retransmitindo texto03.txt a (127.0.0.1:42075)
[Balanceamento] Retransmitindo texto06.txt a (127.0.0.1:42075)
```

supernode.py [2]:

```
Recebendo texto01.txt de (127.0.0.1:36274)
Recebendo texto02.txt de (127.0.0.1:36275)
Recebendo texto03.txt de (127.0.0.1:36276)
Recebendo texto06.txt de (127.0.0.1:36277)
```

supernode.py [3]:

```
[raul@raul-laptop p2p_filesharing]$ python2 supernode.py localhost 6110 /home/raul/estudos/redes/p2p_filesharing/repositorio03/
Recebendo conexões na porta 60456
Superno conectado ao indexador (localhost, 6110) pela porta 38035
```

supernode.py [1]:

```
[Balanceamento] Retransmitindo texto04.txt a (127.0.0.1:60456)
[Balanceamento] Retransmitindo texto05.txt a (127.0.0.1:60456)
```

















supernode.py [2]:

```
[Balanceamento] Retransmitindo texto03.txt a (127.0.0.1:60456)
```

supernode.py [3]:

```
Recebendo texto03.txt de (127.0.0.1:44543)
Recebendo texto04.txt de (127.0.0.1:44548)
Recebendo texto05.txt de (127.0.0.1:44549)
```

Nova estrutura de diretórios após o balanceamento (note que os arquivos não são deslocados por completo, apenas copiados):

- ▼  repositorio01
 -  texto01.txt
 -  texto02.txt
 -  texto03.txt
 -  texto04.txt
 -  texto05.txt
 -  texto06.txt
- ▼  repositorio02
 -  texto01.txt
 -  texto02.txt
 -  texto03.txt
 -  texto06.txt
- ▼  repositorio03
 -  texto03.txt
 -  texto04.txt
 -  texto05.txt

Fazendo requisições a supernós

client.py:

```
[raul@raul-laptop p2p_filesharing]$ python2 client.py localhost 60456 --list
Cliente conectado ao superno (localhost:60456) pela porta 45607
Lista de arquivos disponiveis em (127.0.0.1:60456):
texto03.txt
texto04.txt
texto05.txt
```

Observe a passagem de argumentos:

- --list: operação de listagem
- localhost: hostname do superno
- 60456: porta em que o superno recebe requisições

No caso, localhost:60456 corresponde ao supernode.py [3].

A operação retorna ao cliente os arquivos contidos no repositório de supernode.py [3], no caso o diretório “/home/raul/estudos/redes/p2p_filesharing/repositorio03/”

Observe que assim que a requisição foi atendida o script chegou ao fim, os dois lados da conexão foram fechados. Assim funciona para todas as requisições do cliente.

client.py

```
[raul@raul-laptop p2p_filesharing]$ python2 client.py localhost 60456 --upload /home/raul/aciepe_relogio.png
Cliente conectado ao superno (localhost:60456) pela porta 45775
Transmitindo /home/raul/aciepe_relogio.png a (127.0.0.1:60456)
Sucesso!
```

Requisição de upload para supernode.py [3], observe o parâmetro --upload seguido obrigatoriamente do nome ou caminho completo do arquivo (o caso).

Como não existia um arquivo chamado aciepe_relogio.png no repositório do supernode.py [3] (repositorio03), o arquivo foi transmitido com sucesso.

A mesma operação uma segunda vez vai falhar por essa regra:

client.py

```
[raul@raul-laptop p2p_filesharing]$ python2 client.py localhost 60456 --upload /home/raul/aciepe_relogio.png
Cliente conectado ao superno (localhost:60456) pela porta 45967
Arquivo já existe em (127.0.0.1:60456)
```

Ainda não foi necessário um balanceamento dos arquivos na rede, mas veja:

repositorio01 – 6 arquivos

repositorio02 – 4 arquivos

repositorio03 – 4 arquivos

O algoritmo de consistent hashing garante a boa distribuição, então vou enviar mais um arquivo para o supernode.py [1] (repositorio01) e observar esse efeito:

client.py

```
[raul@raul-laptop p2p_filesharing]$ python2 client.py localhost 33515 --upload /home/raul/non-db.txt
Cliente conectado ao superno (localhost:33515) pela porta 59637
Transmitindo /home/raul/non-db.txt a (127.0.0.1:33515)
Sucesso!
```

supernode.py [1]

```
Recebendo non-db.txt de (127.0.0.1:59637)
[Balanceamento] Retransmitindo non-db.txt a (127.0.0.1:60456)
```

supernode.py [3]

```
Recebendo non-db.txt de (127.0.0.1:47284)
```

Nova estrutura:



Como um cliente pode acessar um arquivo na rede que não está presente no supernó em que está conectado (demonstrando o redirecionamento)

client.py

```
[raul@raul-laptop p2p_filesharing]$ python2 client.py localhost 33515 --download aciepe_relogio.png
Cliente conectado ao superno (localhost:33515) pela porta 60575
aciepe_relogio.png não encontrado em (localhost:33515), redirecionando...
Tentando conectar a (127.0.0.1:60456)
Recebendo aciepe_relogio.png de (127.0.0.1:60456)
```

Note que o supernode.py [1] foi acessado para fazer download do arquivo mas, como não continha o arquivo e não era o responsável pela chave, o pedido foi redirecionado ao supernode.py [3], que transmitiu o arquivo para o cliente.

Quando o arquivo não está na rede

client.py

```
[raul@raul-laptop p2p_filesharing]$ python2 client.py localhost 33515 --download joaozinho.rar
Cliente conectado ao superno (localhost:33515) pela porta 32839
joaozinho.rar não encontrado em (localhost:33515), redirecionando...
Tentando conectar a (127.0.0.1:42075)
joaozinho.rar nao esta disponivel na rede no momento.
```

O supernode.py [3] não detém o arquivo “joaozinho.rar” então redireciona a requisição para o detentor da chave, no caso, supernode.py [2], que por sua vez também não tem posse do arquivo, portanto, o arquivo não existe na rede.