Projeto Final

Distributed Photo Organizer

25 de junho de 2022

Realizado por

103530 André Butuc

103154 João Fonseca

Turma P1

Computação Distribuída 2021/22

Licenciatura em Engenharia Informática

Universidade de Aveiro

Índice

Introdução

Com este trabalho, procurou-se desenvolver um sistema de partilha de imagens *P2P* para ser utilizado por um grupo de amigos. Foi possível garantir a eficiência de armazenamento (as imagens serem únicas e estarem distribuídas pelos *peers*) e uma tolerância a falhas (não serem perdidas imagens no caso de um *peer* se ir a baixo).

Escolhas

Esta rede *P2P* foi desenvolvida usando *Python*, pois tínhamos alguma experiencia prévia nesta linguagem devido aos guiões realizados anteriormente. A comunicação entre *peers* foi feita com recurso a *sockets TCP*, pois assim estaria garantida a integridade dos dados transportados, a ligação entre *peers* seria mantida automaticamente (através dos pacotes *TCP Keepalive*) e, se um *peer* se fosse a baixo, seriam recebidos 0 *bytes* nos outros *peers* (podendo de seguida ser executado em cada *peer* o algoritmo de tolerância a falhas).

Considerando que o protocolo de transporte (*TCP*) funcionaria sem problemas e que o protocolo *P2P* desenvolvido não teria de ser escalável (visto tratar-se de um grupo de amigos), decidiu-se ligar os *peers* numa rede *mesh*, permitindo que todos se conhecessem entre si. Além disso, cada *peer* tem localmente um dicionário com toda a informação sobre os *peers* da rede - identificador, endereço, *hash* das imagens e tamanho da pasta. Esse dicionário é, em todo o momento, igual em todos os *peers*.

Scripts

daemon.py

Executa um *peer* da rede P2P.

```
$ python3 daemon.py images_folder own_port [peertoconnect_port]
```

images_folder	Diretoria que o *peer* deve usar para armazenar as suas imagens.
own_port	Porto pelo qual o *peer* deve ser contactável.
peertoconnect_port	Porto ao qual o *peer* deve estabelecer ligação para se juntar à rede. (se não for fornecido, o *peer* assume que é o único na rede)

client.py

Executa um cliente da rede P2P.

```
$ python3 client.py peertoconnect_port
```

peertoconnect_port	Porto ao qual o cliente deve estabelecer ligação para se ligar a um *peer*.

launch_network.py

Executa automaticamente vários *peers* da rede P2P.

```
$ python3 launch_network.py n_peers
```

n_peers	Número de *peers* que serão executados (3 por omissão).

Protocolo

Identificadores na rede

Para permitir a utilização das mensagens anteriores por *peers* e clientes sem ser necessário haver 2 protocolos diferentes (cliente-peer, peer-peer), foi definido que:

- todos os **clientes** devem utilizar o identificador partilhado `0` nas mensagens, e
- todos os ***peers*** devem utilizar um identificador único `≥1` nas mensagens.

Mensagens

Mensagem	Descrição	Formato	
`JoinMessage`	Enviada por um *peer* que se pretende juntar à rede, para qualquer *peer* que já se encontre na rede.	`{ "command": "join", "addr": tuple }`	
		`addr`	Endereço do *peer* que se pretende juntar à rede.
`ConfigMessage`	Enviada por um *peer* como resposta a uma `JoinMessage`, contém dados para a configuração do *peer* que se juntou à rede.	`{ "command": "config", "from_id": int, "new_id": int, "net_info": dict }`	
		`from_id`	Identificador do remetente.
		`new_id`	Identificador que o recetor deve assumir na rede.
		`net_info`	Dicionário com toda a informação sobre os *peers* da rede – identificador, endereço, *hash* das imagens e tamanho da pasta.
`UpdateMessage`	Enviada por um *peer* para solicitar a um outro que atualize a sua informação sobre a rede com os dados contidos nesta mensagem.	`{ "command": "update", "from_id": int, "add": dict, "rem": dict }`	
		`from_id`	Identificador do remetente.
		`add`	Dicionário com a informação sobre os *peers* da rede que o *peer* deve adicionar/substituir no seu dicionário.
		`rem`	Dicionário com a informação sobre os *peers* da rede que o *peer* deve remover no seu dicionário.
`RequestImageMessage`	Enviada por um *peer* (ou cliente) para solicitar o envio de uma imagem para si.	`{ "command": "request_image", "from_id": int, "hash": bytes }`	
		`from_id`	Identificador do remetente.
		`hash`	*Hash* da imagem.

<table>
<tr><th>Mensagem</th><th>Descrição</th><th colspan="2">Formato</th></tr>
<tr><td rowspan="6">ImageMessage</td><td rowspan="6">Enviada por um peer, contém uma imagem e informação a ela associada.</td><td colspan="2"><pre>{
  "command": "image",
  "from_id": int,
  "hash": bytes,
  "image": Image,
  "fname": str,
  "store": bool
}</pre></td></tr>
<tr><td>from_id</td><td>Identificador do remetente.</td></tr>
<tr><td>hash</td><td>Hash da imagem.</td></tr>
<tr><td>image</td><td>Instância da imagem em PIL.</td></tr>
<tr><td>fname</td><td>Nome do ficheiro da imagem.</td></tr>
<tr><td>store</td><td>Indica se o peer deve ou não guardar a imagem localmente.</td></tr>
<tr><td rowspan="2">RequestListMessage</td><td rowspan="2">Enviada por um cliente para solicitar a um peer o envio de uma lista contendo todos os hashes das imagens na rede.</td><td colspan="2"><pre>{
  "command": "image",
  "from_id": int
}</pre></td></tr>
<tr><td>from_id</td><td>Identificador do remetente.</td></tr>
<tr><td rowspan="2">ListMessage</td><td rowspan="2">Enviada por um peer como resposta a uma RequestListMessage, contém uma lista com todos os hashes das imagens na rede.</td><td colspan="2"><pre>{
  "command": "image",
  "hashes": list
}</pre></td></tr>
<tr><td>hashes</td><td>Lista com todos os hashes das imagens na rede.</td></tr>
</table>

Procedimentos

Identificação unívoca de uma imagem

De forma a garantir que não existem imagens duplicadas na rede, recorreu-se à função **`average_hash()`** da biblioteca *Pillow*. Ao passar uma imagem como argumento dessa função, obtém-se o *hash* dessa imagem. Imagens com o mesmo conteúdo mas com dimensões e/ou cores diferentes irão produzir o mesmo *hash*. Por isso, se duas imagens na mesma pasta produzem o mesmo *hash*, mantém-se aquela que tiver mais informação.

Entrada na rede

Considerando que D1 e D2 são *peers* que formam a rede *P2P*, e que D3 se pretende juntar, se iniciar comunicação com D1:

1. D3 envia uma **`JoinMessage`** para D1, incluindo o seu endereço pelo qual é contactável;
2. D1 recebe uma **`JoinMessage`** de D3,
    a. Determina um identificador único na rede que D3 deve assumir;
    b. Atualiza a sua informação local (endereço de D3 e ligação vinda de D3);
    c. Envia uma **`ConfigMessage`** para D3, incluindo o identificador que esse deve assumir e o dicionário com toda a informação sobre os *peers* da rede.
    d. Envia uma **`UpdateMessage`** para todos os *peers* da rede, exceto o D3 e ele próprio (neste caso, seria apenas para D2), solicitando a adição do endereço de D3 nos dicionários locais.
    e. **`(I)`** Se D3 fosse o segundo *peer* a juntar-se à rede (neste caso, seria o terceiro), D1 enviar-lhe-ia várias **`ImageMessage`**, cada uma com uma cópia de cada imagem que armazenasse localmente.
3. D3 recebe uma **`ConfigMessage`** de D1,
    a. Assume o identificador que lhe foi atribuído;
    b. Substitui o seu dicionário local (até então vazio) pelo que lhe foi enviado;
    c. Analisa a pasta com as suas imagens, removendo imagens que sejam duplicadas localmente e que já estejam na rede (pois agora tem conhecimento da rede);
    d. Envia uma **`UpdateMessage`** para todos os *peers* da rede, exceto ele próprio (neste caso, seria para D1 e D2), solicitando a adição de todos os seus códigos *hash* e tamanho da sua pasta nos dicionários locais;
    e. **`(I)`** Envia uma **`ImageMessage`** para todos os *peers* da rede de forma circular, exceto ele próprio (neste caso, seria para D1 e D2), solicitando o armazenamento de uma imagem que este possua localmente (e que neste momento é única em toda a rede);
4. D1 e D2 recebem uma **`UpdateMessage`** de D3
    a. Atualizam a sua informação local com os códigos *hash* e tamanho da pasta de D3;
    b. Se esses *peers* ainda não estabeleceram uma ligação com D3 (neste caso, seria apenas D2), então enviam uma **`UpdateMessage`** vazia, apenas para que D3 registe as ligações vindas de cada *peer*, e assim saber quando um deles se vai a baixo.
5. **`(I)`** D1 e D2 recebem várias **`ImageMessage`** de D3
    a. Atualizam a sua informação local (*hash* da imagem e guardam a imagem);
    b. Enviam uma **`UpdateMessage`** para todos os *peers* da rede, exceto a eles próprios, solicitando a adição deste código *hash* e tamanho da sua pasta nos dicionários locais;

`(I)` Pertence ao processo de replicação de imagens.

Transferência de uma imagem

Considerando que D1 e D2 são *peers* da rede, e que D2 deve enviar a D1 uma imagem:

1. D1 envia uma **`RequestImageMessage`** para D2, solicitando o envio da imagem correspondente ao código *hash* fornecido. Esta mensagem pode ser enviada em duas situações:
    a. Um cliente solicitou uma imagem que não está armazenada em D1, então o *hash* da imagem é associado ao pedido do cliente e guardado para um posterior encaminhamento.
    b. **`(I)`** Um dos *peers* foi-se a baixo e D1 deve armazenar uma cópia de uma das suas imagens, então o *hash* da imagem é associado a um pedido do próprio e guardado para um posterior armazenamento.
    Nota: Como cada *peer* sabe em que *peers* estão armazenadas as imagens, o pedido é feito diretamente para um dos *peers* que a armazenar, não sendo necessário encaminhamentos desta mensagem entre *peers*.
2. D2 recebe uma **`RequestImageMessage`** de D1,
    a. D2 envia uma **`ImageMessage`** para D1, com a imagem solicitada.
3. D1 recebe uma **`ImageMessage`** de D2,
    a. Se o *hash* estiver em pedidos de clientes, a imagem é encaminhada para esses clientes.
    b. **`(I)`** Se o *hash* estiver nos pedidos do próprio *peer* ou estiver sinalizada para ser armazenada, a imagem é armazenada no próprio.
        i. D1 envia uma **`UpdateMessage`** para todos os *peers* da rede, exceto ele próprio, solicitando a adição deste código *hash* e tamanho da sua pasta nos dicionários locais.

`(I)` Pertence ao processo de replicação de imagens.

Encaminhamento de uma imagem

O encaminhamento de uma imagem só existe caso um cliente peça uma imagem que não se encontra armazenada no *peer* ao qual está ligado.

Considerando D1, D2 e um cliente que está ligado a D1:

1. O cliente envia uma **`RequestImageMessage`** a D1.
2. D1 recebe uma **`RequestImageMessage`** do cliente,
    a. Como sabe que veio do cliente que se encontra ligado a ele (pelo facto de ter como identificador o valor **`0`**), verifica se possui ou não o *hash* solicitado.
    b. Determina que o *hash* encontra-se em D2, então o *hash* da imagem é associado ao pedido do cliente e guardado para um posterior encaminhamento, após receber uma **`ImageMessage`** com esse *hash*;
    c. D1 envia uma **`RequestImageMessage`** para D2 com o *hash* da imagem pretendida.
3. D2 recebe uma **`RequestImageMessage`** de D1,
    a. D2 envia uma **`ImageMessage`** para D1, com a imagem solicitada.
4. D1 recebe uma **`ImageMessage`** de D2,
    a. Como o *hash* se encontrava nos pedidos do cliente, D1 envia uma **`ImageMessage`** ao cliente com a imagem solicitada.

Crash de um *peer*

Considerando que D1, D2, D3 e D4 são *peers* que formam a rede *P2P*, caso D3 falhe, D1, D2 e D4 iniciam o algoritmo de tolerância a falhas de forma a assegurar que as imagens que estavam associadas a D3 continuem a estar duplicadas na rede.

Quando D3 falha, as conexões *TCP* estabelecidas com os outros três *peers* (D1, D2 e D4) recebem 0 *bytes*, pelo que os *peers* sabem que devem iniciar o algoritmo de tolerância a falhas. Este algoritmo deve ter o mesmo comportamento em todos os *peers*, visto que todos possuem a mesma informação sobre a rede, e funciona da seguinte forma:

1. D1, D2 e D4 fazem o backup dos códigos *hash* de D3, removendo-os do dicionário local da morfologia da rede.
2. D1, D2 e D4 elegem um *peer* designado e um *peer* substituto, que irão assumir a responsabilidade de replicar as cópias perdidas de D3. A eleição é feita tendo em conta dois critérios: menor tamanho da pasta do *peer* e, em caso de empate, o *peer* com menor valor do identificador. Por questões de simplificação, iremos assumir que o *peer* designado foi o D1 e o *peer* substituto foi o D4. Todos os *peers* não eleitos terminam o seu algoritmo de tolerância a falhas.
3. D1 itera o backup dos códigos *hash* (feito no passo 1) e procura o identificador do *peer* que tem em sua posse cada *hash*. Isto é feito até todas os códigos *hash* de D3 serem percorridos:
    a. Se o identificador obtido for igual ao identificador de D1, isto é, que D1 tem a cópia da imagem de D3, D1 envia uma **`ImageMessage`** com o campo **`store`** igual a **`True`** para o *peer* substituto, D4. Esse recebe a **`ImageMessage`** de D1 e sabe que deve armazenar a imagem que lhe está associada.
    b. Se o identificador obtido não for igual ao identificador de D1, então o *hash* da imagem é associado a um pedido do próprio e guardado para um posterior armazenamento. D1 envia uma **`RequestImageMessage`** ao *peer* que tem na sua posse o *hash* da imagem pretendida.

O *peer* D4 irá receber várias **`ImageMessage`** de D1 para armazenar localmente, assim como o *peer* D1 irá receber várias **`ImageMessage`** de outros *peers* em resposta às suas **`RequestImageMessage`**. Segundo os procedimentos abordados anteriormente, cada um irá enviar as **`UpdateMessage`** correspondentes para o resto da rede, atualizando cada dicionário local. Assim, a rede volta a manter duas cópias de cada imagem.

Nota final: Este algoritmo de tolerância a falhas visa a tornar o sistema de armazenamento *P2P* redundante, pelo que o processo de replicação de imagens está espelhado no mesmo.

Estabelecimento de ligações, *threads* e *mutexes*

Ao inicializar um *peer*, o seu porto que está em *listen* é registado num *selector* local, que irá executar a função **`accept()`** sempre que houver o estabelecimento de uma nova ligação a esse porto.

Quando uma ligação é estabelecida, é executada a função **`accept()`**, sendo a ligação registada no mesmo selector, que irá executar a função **`read()`** sempre que houver dados para ler dessa ligação.

Quando houver dados para ler dessa ligação, é executada a função **`read()`**, que irá ler os dados recebidos. Se a socket correspondente fechou (foram recebidos 0 *bytes*), poderá ser despoletado o algoritmo de tolerância a falhas (se a ligação era de um *peer*), ou nada (caso fosse de um cliente). Em todo o caso, a ligação é removida do *selector* e fechada. Caso tenha sido recebida uma mensagem válida no protocolo definido, é lançada uma *thread* responsável por processar tal mensagem. Dessa forma, é possível atender a vários pedidos em simultâneo, ao mesmo tempo que a *thread* principal lê dados das ligações.

Sentimos a necessidade de, na nossa solução, implementar *threads*, pois inicialmente fomos deparados com situações em que os *peers* ficavam num estado de *deadlock*. Isso acontecia no caso de um *peer* ficar bloqueado no *send* por o buffer ficar cheio, não podendo receber dados que lhe chegavam, e bloqueando desse modo os outros *peers* nos seus próprios *sends*.

No entanto, como agora tínhamos *threads*, estas podiam escrever em simultâneo na mesma *socket* (no caso de haver múltiplos pedidos direcionados para uma mesma ligação), o que levou a que as mensagens fossem corrompidas. Por isso, houve a necessidade de controlar as escritas nas *sockets* através de um *mutex* (**`_send_mutex`**).

Mais tarde, foi também necessário adicionar mais um *mutex* (**`_folder_mutex`**). Este serve para lidar com escritas concorrenciais na pasta com as imagens de cada *peer*, além da leitura da pasta para cálculo do tamanho das suas imagens, que não poderia ser feito ao mesmo tempo que uma imagem estivesse a ser escrita.