universidade de aveiro deti departamento de electrónica, telecomunicações e informática

# Distributed Photo Organizer

## Projeto Final de Computação Distribuída 2021/22

Prof. Diogo Gomes
Prof. Nuno Lau
Prof. João Rodrigues

**Artur Afonso Ferra Correia (102477)**
**Daniel Luís Fernandes Carvalho (77036)**
Licenciatura em Engenharia Informática
Turma P2

# 1. Estrutura Geral da Rede

A rede peer-to-peer desenvolvida para este projeto está assente na identificação de cada imagem por uma hash, e na distribuição dessas hashes pelos vários nós Daemon da rede, que se devem conhecer mutuamente e manter comunicação entre si através de protocolos de *routing* estabelecidos. Além de cada nó ter conhecimento de todos os outros nós da rede, todos os eles partilham também uma estrutura geral de dados, a keystore, que indica quais os hashes e as imagens armazenadas por cada nó. Deste modo, qualquer nó pode, ao receber um pedido de um cliente para transferir uma imagem especifica, determinar e contactar diretamente o nó onde esta está armazenada.

A escolha desta estrutura para a rede deve-se ao facto de nós sabermos à partida que esta vai ter um número reduzido de nós, e relativamente reduzido de ficheiros partilhados, pelo que o tráfego de mensagens será diminuto e sustentável por uma estrutura menos hierárquica, como a pretendida.

Para o desenvolvimento do código de cada nó Daemon, foi adaptada a estrutura da classe DHTNode desenvolvida para a resolução do Guião 2 sobre o CHORD DHT, uma vez que a rede aí desenvolvida era também peer-to-peer e trabalhava também com uma distributed hash table. Estas semelhanças permitiram o reaproveitamento da estratégia de envio e receção de mensagens entre as sockets de cada nó assente no protocolo UDP, e com base no pacote Pickle, sendo considerado que para uma rede de dimensão relativamente baixa como a pretendida, e na qual o tipo de mensagens trocado é limitado, a utilização deste protocolo será eficiente. No entanto, a diferença estrutural entre as duas redes implicou a readaptação de grande parte da classe, em particular nos processos de entrada de novos nós na rede e da estabilização e manutenção das tabelas de endereçamento dos nós, que serão discutidas mais à frente. Além disso, foi desenvolvido todo um novo protocolo para o tipo de mensagens enviadas, estando este descrito no documento “protocol.txt”.

# 2. Protocolo de Estabilização da Rede

## 2.1. Inicialização da Rede e Entrada de Novos Nós

De acordo com o indicado no README.md presente na entrega, para se lançar um novo nó Daemon através do terminal é necessário passar-lhe como parâmetro o seu ID e o seu endereço. Caso se pretenda lançar o primeiro nó que vai estabelecer a rede, não é necessário passar mais nenhuma informação, com o nó a ancorar a nova rede criada; no entanto, se se pretende que o novo nó lançado seja associado a uma rede previamente existente, é necessário passar também

como argumento o endereço de um nó presente nesta (denominado daqui em diante como nó contacto).

Para um nó juntar-se a uma rede, é enviado uma mensagem JOIN_REQUEST ao nó contacto, que vai responder com uma JOIN_REPLY, na qual vai incluir a sua tabela de endereçamento atual, bem como a cópia da keystore. Não é necessário que o nó contacto seja o nó original que estabeleceu a rede, pelo que qualquer nó pode servir como ponto de entrada para um novo elemento.

Deste modo, o novo nó vai logo à partida tomar conhecimento de todos os nós existentes na rede, bem como de todas as imagens guardadas por esta. Após a entrada, o novo nó contacta os restantes nós da rede com uma mensagem do tipo HELLO, na qual indica as imagens por si guardadas, de maneira que os seus peers possam tomar conhecimento da sua entrada na rede, e registar a existência e a localização das imagens por si guardadas na sua keystore. É também imediatamente acionado o mecanismo de backup de imagens, explicado na secção 4.

## 2.2. Protocolo de Estabilização

A tabela de endereçamento anteriormente mencionada é essencial para o mecanismo de estabilização da rede. Cada nó presente na rede vai ter uma tabela deste tipo, onde estão listados todos os nós vizinhos com quem ele tem contacto, o seu routingStatus, e a timestamp da última vez que houve comunicação com este nó, tendo sido definido que o routingStatus pode ser 1 de 4 estados: ALIVE, CHECKING, SUS ou DEAD. Deste modo, sempre que um nó comunica com outro, vai registar na sua tabela de endereçamento o estado deste como ALIVE, e atualizar o timestamp de último contacto.

Quando um nó está inativo, sem realizar qualquer operação, vai eventualmente ocorrer um timeout, onde é iniciado o processo de estabilização. Neste processo, o nó vai percorrer a sua tabela de endereçamento e calcular para cada nó vizinho quanto tempo passou desde o último contacto. Se o intervalo de tempo para um determinado nó for maior do que o tempo KEEPALIVE (definido como 30 segundos), o estado deste nó é alterado para CHECKING, e é enviada a ele uma mensagem ALIVE. Caso o nó esteja vivo, responde com uma mensagem ALIVE_ACK, e o seu estado é revertido de volta para ALIVE. Pelo contrário, caso o nó em questão não responda nem estabeleça comunicação dentro do próximo ciclo de atividade, o seu estado passa para SUS, com a inatividade em mais um ciclo de timeout a resultar na passagem para o estado DEAD, e na ativação do mecanismo de salvaguarda de qualquer backup seu armazenado pelo nosso nó, descrito na secção 4.

Inicialmente, todos os nós contactavam os seus vizinhos no final de um timeout com um pedido de ALIVE, sendo que o estado de cada nó era apenas dois: ALIVE e DEAD. No entanto, verificou-se que isto resultava num rápido congestionamento da rede quando se aumenta o

número de nós presentes, em particular quando se define valores de timeout baixos (inferiores a 10-15 segundos), o que resultava na eventual troca das mensagens esperadas e na falha do desempacotamento destas pelo Pickle. Assim, foi decidido estabelecer-se o mecanismo de salvaguarda do timestamp do último contacto e da comparação deste com o tempo KEEPALIVE, o que evitou a necessidade de envio de mensagens ALIVE a todos os nós num timeout, diminuindo consideravelmente o congestionamento da rede.

Além disso, a adição dos estados CHECKING e SUS permite também garantir que um nó que tenha mal funcionado temporariamente, ou que tenha estado ocupado a responder a outros pedidos, tenha chance de se reconectar à rede, sem ser imediatamente eliminado das tabelas de endereçamento dos seus vizinhos, e impedindo que estes transfiram desnecessariamente os backups que estavam a salvaguardar nesse nó para outros vizinhos.

É de notar que a escolha da definição do tempo KEEPALIVE como 30 segundos foi feita com base em várias experiências, nas quais se consideraram diferentes combinações de tempos KEEPALIVE e timeout, tendo se determinado que este valor permite a manutenção de uma rede estável sem grandes problemas de congestionamento com pacotes ALIVE e ALIVE_ACK.

As funções implementadas para suportar o mecanismo descrito foram as seguintes:

- **stabilize:** Define os passos do protocolo de estabilização
- **stay_alive:** Percorre a tabela de endereçamento do nó e envia uma mensagem ALIVE aos vizinhos cujo último contacto tenha sido realizado num intervalo de tempo superior ao tempo KEEPALIVE
- **check_alive:** Caso um nó previamente contactado por uma mensagem ALIVE não tenha respondido, o seu estado na tabela de endereçamento é alterado (de CHECKING para SUS e de SUS para DEAD)

# 3. Identificação, Pesquisa e Transferência de Imagens na Rede

Para a implementação da troca de imagens foi utilizada a biblioteca PIL, recomendada pelo docente, que possibilitou abrir as imagens como objetos PIL e usar esses objetos diretamente nas mensagens enviadas. Para além disso, também se recorreu à biblioteca ImageHash, que converte uma imagem guardada (objeto PIL) numa hash, que será posteriormente utilizada para a comparação das imagens na rede criada, garantindo que não estão repetidas, excetuando os backups feitos.

Foi utilizado o algoritmo Difference hashing (dhash) nas imagens, sendo que esta escolha foi feita com basa na documentação fornecida pela biblioteca ImageHash, mais concretamente devido à sua performance em relação ao phash, que foi considerado inicialmente por apresentar uma precisão superior comparativamente aos outros, contudo após a sua utilização notou-se que este não estava a obter o comportamento desejado. Por fim, foi, também, comparado os Art

Datasets de todos os algoritmos e identificamos que para as imagens disponibilizadas o dhash seria o mais correto.

Ocorrem transferências de imagem quando existe um pedido de imagem por parte de um cliente, ou, então, quando algum nó está a fazer um backup, sendo que o segundo caso apenas será explicado na parte de Backups do relatório, secção 4. Quando um cliente pede uma imagem, através da mensagem REQUEST_IMAGE, o nó de contacto com o cliente irá verificar na sua Keystore se possui essa imagem, esta pesquisa é feita através do nome da imagem. Se o nó possui a imagem pedida irá enviar ao cliente através de uma REPLY_IMG, contudo se verificar que não é o nó que guarda a imagem irá enviar uma mensagem REQUEST_IMAGE ao nó que a estiver a guardar. Esse nó posteriormente irá enviar a imagem ao cliente que a pediu diretamente, através de uma REPLY_IMG.

Para que esta troca de mensagens fosse possível foram criadas algumas funções para ajudar na implementação:

- **get:** para verificar se qual dos nós possui a imagem requisitada
- **get_key:** para auxiliar o get a encontrar o nó
- **send_image :** para compor a mensagem REPLY_IMG e chamar a função **send**

Para enviar uma mensagem com uma imagem foi necessário efetuar algumas mudanças na função **send**, pois não é possível enviar uma mensagem superior a 4096 bytes, o que levou a dividir a mensagem em partes que serão enviadas sequencialmente. Posteriormente, no **recv** foi necessário reconstruir a mensagem, juntando as partes da mensagem à medida que se recebiam. Para o efeito, foi assumido que as mensagens iriam ser enviadas de forma sequencial. Apesar de não existir essa garantia, após alguns testes e colocando um intervalo entre o envio das mensagens, conclui-se que as mensagens foram reconstruídas de forma bem-sucedida e que, de facto, o envio foi sequencial como esperado. Contudo no que toca à escalabilidade da rede é notório que deveria ser incluído um protocolo diferente para o envio das imagens. Uma sugestão dessa implementação seria uma mensagem que possui como argumento o número da parte da imagem, que depois ao receber seria guardada numa estrutura e quando se reconstruisse a mensagem organizando as partes pela ordem correta.

# 4. Protocolo de Tolerância a Falhas

Para garantir que o sistema é robusto e permite uma adaptação dinâmica a acontecimentos imprevisíveis da rede foi criado todo um conjunto de mensagens e métodos de Backup.

Foi atribuído a cada nó o comportamento de efetuar os seus backups caso entre na rede e já tenha um peer, ou mais, ou então quando acontece um timeout e este verifica que nem todas as suas mensagens possuem um backup na rede. Para isso foram implementadas várias funções:

- **check_backedup_images:** para verificar se as imagens já possuem backups
- **set_backups:** para distribuir as imagens que cada peer irá guardar backup
- **send_backup:** para preparar o envio das mensagens de backup

- **receive_backup:** para receber e tratar dos backups

A escolha do timing de Backup foi decidida com base na estrutura da rede, sendo que faria sentido o nó ao se juntar e organizar a sua pasta de imagens, efetuar imediatamente um backup com os seus peers, ou quando ocorre um timeout e ele verifica que nem todas as suas imagens possuem backup, pois são momentos em que o nó não está a tratar da receção de mensagens, apenas se estiver a estabilizar. Tal possibilita realizar os backups, um processo relativamente longo (3-6s), sem que existam perdas de informação por overlap de mensagens, ou então entupimentos na socket.

Quando um nó se desliga da rede, por ter sido desconectado, ou por algum erro no processamento de mensagens, os peers dele ao efetuarem o processo de estabilização irão alterar o estado desse nó. Quando um nó é declarado como morto, DEAD, é iniciado o processo de recuperação de imagens. Cada nó que possuir um Backup do nó morto irá passar as imagens armazenadas na pasta de Backup do nó para a pasta principal e posteriormente irá apagar a pasta de Backup desse nó. As cópias dessas mensagens irão ser efetuadas novamente, pois ao efetuar um backup ele irá detetar que elas ainda n têm um backup na rede e serão criados novos backups para as imagens. Conclui-se, que nunca irão existir perdas de informação, ou seja, as imagens estarão sempre na rede.