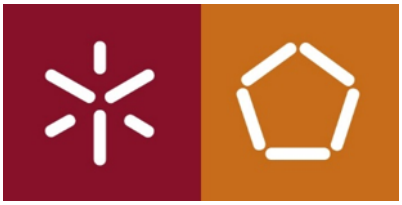

UNIVERSIDADE DO MINHO

SISTEMAS DISTRIBUÍDOS EM LARGA ESCALA

Decentralized Timeline

RELATÓRIO DE DESENVOLVIMENTO

Mestrado Integrado em Engenharia Informática

Realizado pelo grupo G:
Hugo André Coelho Cardoso, a85006
José Pedro Oliveira Silva, a84577
Pedro Miguel Borges Rodrigues, a84783
Válter Ferreira Picas Carvalho, a84464

9 de junho de 2021

Conteúdo

1 Introdução **2**

2 Arquitetura **3**
2.1 Escolha de Tecnologias . . . 3
2.2 Arquitetura da Rede . . . 3
2.3 Arquitetura de um *Peer* . . . 5

3 Solução Final **6**
3.1 Funcionalidades . . . 6
3.2 Implementação . . . 6
3.2.1 Ordenação Causal . . . 7
3.2.2 *Queue* de mensagens assíncronas . . . 7
3.2.3 Estabelecimento de conexões TCP . . . 7
3.2.4 Política de descarte de mensagens . . . 8

4 Conclusão **9**

Lista de Figuras

1 Arquitetura da rede P2P usando Kademlia . . . 4
2 Arquitetura de um *Peer* . . . 5

1 Introdução

No âmbito da Unidade Curricular *Sistemas Distribuídos em Larga Escala*, inserida no perfil de mestrado de Sistemas Distribuídos, lecionado no Mestrado Integrado em Engenharia Informática da Universidade do Minho, foi proposta a resolução do projeto prático **Decentralized Timeline**.

A premissa deste projeto é a criação de uma *timeline* (semelhante ao Twitter e Instagram) mas tendo em conta uma rede *peer-to-peer* (P2P). Cada *peer* pode escrever uma mensagem que aparecerá no seu *feed* e na dos *peers* que lhe decidiram subscrever.

A solução obtida tira partido das ferramentas lecionadas e utilizadas em contexto de aula, nomeadamente a utilização de **Python**, que foi a linguagem de programação eleita desde o início do semestre até para efeitos de aulas, e a *framework* **Kademlia DHT** para simular a rede P2P e os nodos que existem nesse sistema. Para além destas, o grupo procurou utilizar ferramentas novas complementares, como é o caso do **MongoDB**, para armazenamento persistente, e o respetivo controlador **PyMongo**; **BCrypt** para encriptação de palavras-chave e **AsyncIO**, que permite a utilização do paradigma de programação orientada a eventos em Python.

De seguida será explicitado a arquitetura utilizada, assim como a justificação para todas as escolhas feitas pelos elementos do grupo, de modo a atingir a solução final, assim como os problemas que surgiram durante o desenvolvimento da aplicação.

2 Arquitetura

2.1 Escolha de Tecnologias

O projeto foi construído de raíz utilizando a linguagem de programação **Python**, dado que é uma linguagem muito usada a nível mundial devido à sua eficácia e simplicidade e, para este projeto em particular, fornecia todas as *frameworks* necessárias para a sua implementação. Uma das particularidades desta linguagem é a facilidade de tradução entre documentos **JSON** e dicionários nativos, pelo que influenciou bastante a escolha do método de armazenamento persistente.

Uma destas *frameworks* em particular é a implementação da tabela de *hash* distribuída (DHT) **Kademlia** (`https://pypi.org/project/kademlia/`), que abstrai e encapsula o funcionamento da comunicação entre os nodos participantes para uma rede *peer-to-peer*. As principais vantagens desta implementação em concreto acima de outros tipos de DHTs são a eficiência nos *lookups*, que garante complexidade logarítmica no pior caso e a métrica de distância XOR utilizada pelo algoritmo, que garante a simetria nos roteamentos, isto é, se A está na tabela de roteamento de B, então B está na tabela de roteamento de A, o que permite cálculos de rotas alternativas de forma muito mais eficiente.

A *framework* descrita tira partido de programação orientada a eventos através do módulo **AsyncIO**, que gere uma *queue* de tarefas executadas assincronamente, tais como leitura e escrita em *sockets*, de modo a eliminar a existência de métodos bloqueantes. Por estes motivos de eficiência e por consistência, todo o comportamento do programa será assíncrono, garantindo assim que para qualquer operação executada num *peer*, ela nunca bloqueará a *thread* em que o *event handler* está a atuar, que é o principal objetivo deste paradigma.

A aplicação fornece a possibilidade do *peer* se autenticar na rede. Como a tabela é distribuída pelo vários nodos no sistema, a informação crítica de um *peer* - neste caso a palavra-passe - pode não ficar nele mesmo, devido à natureza do *hashing* efetuado pelo Kademlia para guardar um par chave-valor. Logo, esta tem de ser encriptada de forma assimétrica, isto é, não deve ser possível obter a palavra-passe original mesmo sabendo o algoritmo de encriptação. Para tal, foi utilizado o módulo **BCrypt**, que faz exatamente este processo, assim como a validação de palavras-passe tendo em conta o *hash* das mesmas.

A persistência de dados é obtida através de **MongoDB**, uma base de dados não relacional do tipo *document-store*, que em particular guarda em cada registo um objeto JSON. Assim, utilizando Python, é possível traduzir diretamente um objeto persistente para memória volátil, como já foi mencionado anteriormente. Esta escolha implica utilizar o controlador **PyMongo**, que encapsula os métodos de acesso ao armazenamento em funções de alto nível em Python.

2.2 Arquitetura da Rede

Como mencionado anteriormente, a rede baseia-se num sistema *peer-to-peer* utilizando Kademlia. Cada nodo, portanto, tem uma tabela de *hash* única que é acedida quando há operações de **GET** ou **PUT** nas mesmas por parte de um nodo remoto, quando o *hash* calculado sobre uma chave aponta para esse nodo que a contém.

Com esta implementação é garantido que a informação sobre um determinado nodo está sempre disponível uma vez que, em caso de falha de um nodo, a sua tabela é transferida para um nodo operacional, que é uma das funcionalidades registadas pela *framework* que está a ser utilizada.

Garante, assim, a tolerância a faltas uma vez que, para garantir que o sistema colapsa totalmente, os N *peers* participantes têm de falhar **simultaneamente** de modo a não conseguirem executar a transferência de tabelas, o que é uma situação muito rara em sistemas com possivelmente milhões de utilizadores. Para além disso, a falha de um nodo individual não causa falhas no sistema, pelo que é totalmente escalável (é uma das características desejáveis das redes P2P).

Esta tabela, em cada uma das suas entradas, mantém guardado o estado de um nodo em qualquer instante de tempo, que está visível na figura seguinte.

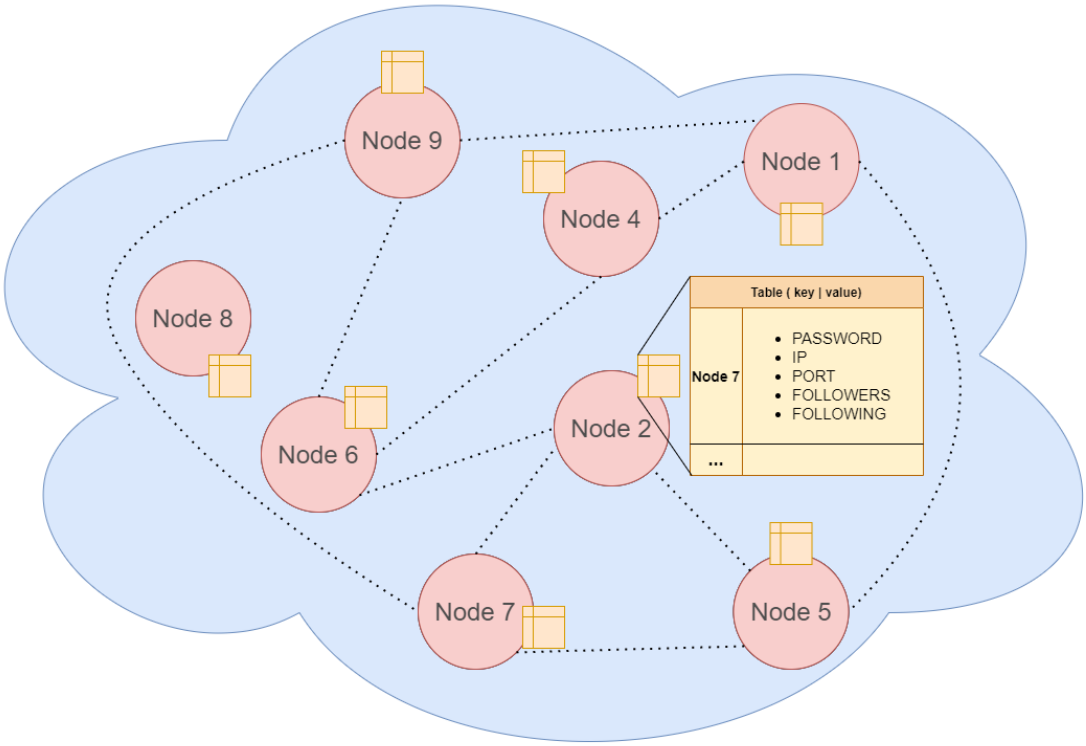

Figura 1: Arquitetura da rede P2P usando Kademlia

Para cada par chave-valor, o **nome** de utilizador (representa um *peer* no sistema) atua como chave e o valor é um dicionário (equivalente a JSON) com os seguintes campos:

- **PASSWORD**: Contém o resultado de aplicar o algoritmo de BCrypt à *password* do utilizador aquando o registo no sistema. Este *hash* garante que o nodo que contenha a informação deste nodo registado não consiga decifrar a palavra-passe visto que não é simétrico. Exemplo: *"$2b$06$f.x1ftoNYtauKyaXMYpTSOvBO5il/b5olbGLN/EgbJ3OsCQqJDimG"*
- **IP**: Refere-se ao endereço IPv4 do *peer*, que pode ser local ou numa outra rede. Este IP será utilizado para realizar comunicação com um outro nodo, caso seja necessário. Este tipo de comunicação é realizado sobre *sockets* TCP e é o canal dedicado, entre outros tipos de mensagens a serem vistas mais à frente, à transmissão de *posts* na *timeline* de um *peer* para os seus subscritores. Exemplos: *"localhost", "63.91.219.240"*
- **PORT**: Para além do IP, é necessário a porta para executar comunicação através de *sockets* TCP. Assim, cada *peer* tem um porta sempre ativa dedicada à escuta e envio de mensagens relativamente a outros *peers*. Exemplo: *6799*
- **FOLLOWERS**: Refere-se à lista de utilizadores que seguem o *peer* que é atualizada consoante os pedidos de subscrição por outros *peers*. Esta lista é utilizada para o nodo conseguir identificar quais os canais TCP tem de utilizar para disseminar as mensagens. Exemplo: *["john","peter"]*
- **FOLLOWING**: É a implementação de *vector clocks* que garantem a causalidade na entrega de mensagens. Para cada um dos *peers* que um dado *peer* segue, é mantido no relógio lógico o identificador do último *post* que recebeu. É implementado utilizando um dicionário devido ao seu tamanho dinâmico, a utilização de um *array* implicaria conhecer os nodos todos do sistema, o que não é possível. Exemplo: *{"james“ -> 3, "andrew“ -> 2}*

2.3 Arquitetura de um *Peer*

A solução final para arquitetura de um *peer* está presente na imagem abaixo.

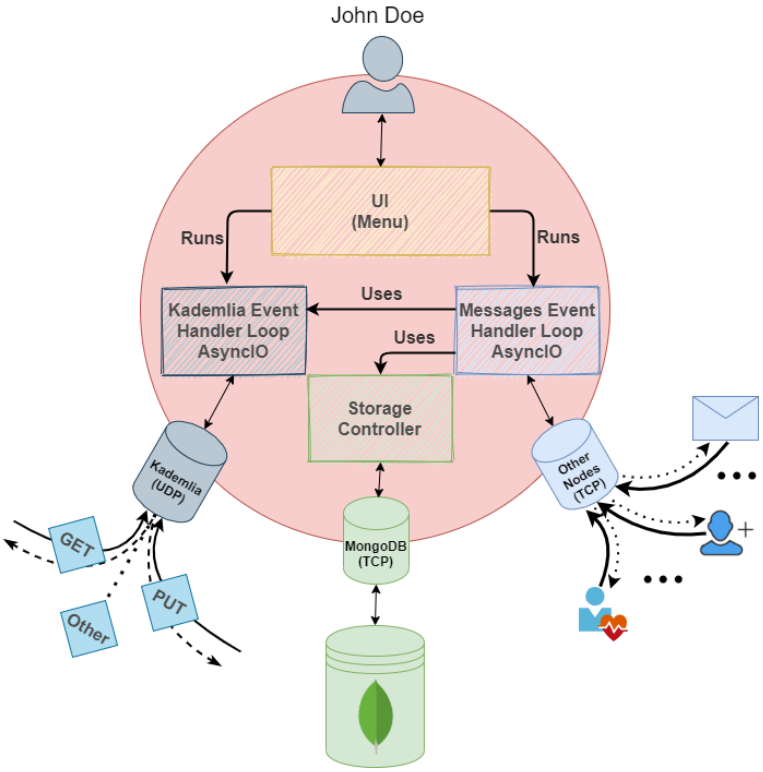

Figura 2: Arquitetura de um *Peer*

Existem, em cada nodo, três portas distintas dedicadas a diferentes tipos de comunicação:

1. **Kademlia (UDP)** - esta porta é dedicada exclusivamente a tráfego utilizado pela *framework* de Kademlia. A ferramenta recebe, essencialmente, tráfego de **GET** e **PUT**, que são as operações sobre as tabelas de *hash* distribuída e, também, outros tipos de mensagens tais como controlo de congestionamento, verificações de integridade, entre outros. A *framework* apenas disponibiliza formas de utilizar este canal de comunicação com estas primitivas e não são configuráveis pelo programador.

2. **MongoDB (TCP)** - tal como foi mencionado anteriormente, para armazenamento persistente do estado de um *peer* é utilizada a base de dados MongoDB, que fornece métodos para ser acedida utilizando TCP, visto que é uma aplicação isolada.

3. **Outros Nodos (TCP)** - na secção anterior foi revelado que, caso seja necessário um *peer* entrar em contacto direto com outro, terá de utilizar o IP e a porta do mesmo para efetuar uma ligação TCP através de *sockets*. Esta porta é utilizada para receber *posts* de um *peer* que está subscrito, de verificação sobre o seu *status* (online ou offline), pedidos de subscrição, remoção de subscrição ou pedido de mensagens em atraso.

Com estes três locais dedicados à comunicação, era necessário ter um leitor e um escritor em cada um destes *sockets*, à exceção do MongoDB que é acedido diretamente utilizando **PyMongo**, tal como explicado anteriormente. Para isso foi utilizado o **AsyncIO** tal como visto na secção inicial, que garante a utilização de uma *queue* de tarefas assíncronas na *thread* com o *event handler* de modo a evitar que operações bloqueantes (escrita e leitura nos *sockets*) não congestionem a performance global do programa.

O utilizador comunica apenas com a interface e dependendo da operação, estas utilizam os canais necessários para atingir o resultado da mesma, o que será explorado na secção seguinte.

3 Solução Final

Neste capítulo, será explicada a solução final desenvolvida com a arquitetura e tecnologias mencionadas anteriormente, endereçando tanto as funcionalidades disponíveis no programa como os problemas que surgiram durante o desenvolvimento do trabalho durante a sua implementação, que servem tanto de causa como de motivação para a solução final.

3.1 Funcionalidades

O programa apresenta uma interface por texto da qual o utilizador pode selecionar a opção que pretende através de *inputs* numéricos. As funcionalidades estão divididas em dois menus - antes e depois do *login*. Procurar-se-á explicar o fluxo da aplicação no curso destas funcionalidades com recurso à figura 3.

Antes da autenticação, é possível o utilizador fazer:

- ***login*** - entrar numa conta já existente, cujos dados estão guardados em memória persistente. No contexto da figura 3, o programa acede ao *Kademlia loop* para fazer a autenticação com os dados na tabela de *hash* distribuída e recorre também ao *Messages loop* para informar os nodos que lhe estão subscritos de que está *online*;
- ***logout*** - terminar a sessão numa conta. O fluxo da aplicação é o mesmo que no *login*, mas o efeito é o contrário - informar os subscritores de que já não está disponível;
- **registo** - registar um novo utilizador (nodo) na rede, associando-lhe um nome de utilizador e uma password. Nesta operação, o programa apenas comunica com o *Kademlia Event Handler Loop* para atualizar as tabelas de *hashing* com os dados novos, não sendo necessário interagir com o *Messages Loop*.

Uma vez autenticado, o utilizador pode realmente usufruir da aplicação. Tem ao seu dispor um conjunto de funcionalidades:

- **subscrever/remover subscrição** a um nodo - (deixar de) seguir um certo utilizador na rede, recebendo (ou deixando de receber) todas as suas mensagens na *timeline*. Em termos de fluxo, a aplicação comunica com o *Kademlia* para atualizar os seguidores (*followers*) dos nodos, adicionando/removendo o nodo em questão, e com os outros nodos, através do *Messages loop*, para notificar os *sockets* a serem ligados;
- **publicar mensagem** - que poderá ser visualizada pelo próprio nodo e pelos seus subscritores. O programa comunica com os restantes nodos de forma a atualizar as *timelines* dos subscritores e com o *Kademlia* para atualizar os *vector clocks* de todos os nodos que recebem a nova mensagem;
- **consultar *timeline*** - permite ver o registo de todas as mensagens (tanto do próprio nodo como daqueles a que está subscrito) que estejam persistidas em memória (as mensagens são efémeras, como será explicado na próxima secção). Esta operação comunica com o *Messages loop* para aceder ao *storage controller*, de forma a ir buscar as mensagens à base de dados.

3.2 Implementação

Ao longo do desenvolvimento do projeto, o grupo deparou-se com vários desafios que moldaram a solução final e exigiram bastante pesquisa e ponderação, em busca da melhor abordagem possível. De seguida, serão abordados esses mesmos desafios, que se traduzem em propriedades do sistema, bem como a abordagem tomada para as garantir:

3.2.1 Ordenação Causal

A ordenação causal é uma propriedade fulcral em sistemas distribuídos, que permite garantir a ordenação de mensagens entre os vários nodos da rede, abstraindo qualquer noção de tempo real ou absoluto - que não é viável, visto que cada nodo tem o seu tempo local, que pode não estar sincronizado com os restantes devido a desvios -, procurando assegurar que as mensagens são recebidas por ordem causal. Isto é, se uma mensagem B é causada/influenciada por uma mensagem A, diz-se que A é causalmente precedente de B e, como tal, todos os nodos devem receber A e só depois B.

Para a implementação deste conceito, uma primeira abordagem consistiu na implementação de relógios lógicos. Contudo, após alguma ponderação, determinou-se que esta solução não seria viável, uma vez que exigiria tentativas de sincronização global constantes entre os nodos, de forma a garantir a ordem local. Seria necessário circular mensagens de controlo pela rede periodicamente entre todos os *peers*, de forma a ajustar os desvios dos relógios, o que não seria nada escalável e rapidamente sobrecarregaria a rede.

Por fim, o grupo recorreu a *vector clocks* - estruturas de dados em que cada nodo guarda o número da última mensagem que recebeu de cada *peer* que segue. Estes relógios dizem respeito ao campo *Following* da DHT de cada nodo e, através deles, é possível garantir que as mensagens são recebidas por ordem causal.

3.2.2 *Queue* de mensagens assíncronas

A aplicação implementa persistência de dados em *MongoDB*, onde são criadas duas coleções por nodo - a *timeline* atual do utilizador e uma *queue* de mensagens cuja receção está pendente. Esta *queue* tem dois objetivos:

- complementar os *vector clocks* no estabelecimento de ordenação causal. Por exemplo, se um nodo tiver o relógio vetorial de um *peer* a 1 e receber a mensagem 3 desse *peer*, é sinal de que ainda falta receber uma mensagem causalmente precedente a esta (2). De forma a não descartar a mensagem 3 e obrigar à sua retransmissão, a mesma é guardada na *queue* em *MongoDB*, e só é removida de lá quando o nodo receber a mensagem imediatamente anterior.

- garantir que os subscritores de um nodo recebem sempre as suas mensagens, mesmo que não estejam na rede (*offline*). Imagine-se o caso em que um nodo A subscreve a um nodo B e dá *logout*. As mensagens que B publicar enquanto A estiver *offline* são armazenadas persistentemente na *queue* de A, na base de dados, de forma a serem apresentadas ao utilizador da próxima vez que este voltar a entrar na rede.

3.2.3 Estabelecimento de conexões TCP

Relativamente a esta questão, o grupo considerou que havia duas implementações possíveis: abrir as conexões TCP todas ao iniciar a aplicação ou abrir cada conexão apenas quando fosse preciso enviar uma mensagem através da mesma.

O estabelecimento de uma conexão TCP é um processo bastante exigente em termos de computação, uma vez que se realiza o *3 way handshake* inicial entre os dois nodos da ligação. Este procedimento, repetido para todas as mensagens enviadas, acabaria por rapidamente sobrecarregar a rede com mensagens de controlo, o que não é escalável nem faz grande sentido num sistema distribuído onde se espera haver um grande fluxo de mensagens entre os nodos.

Desta forma, o grupo acabou por adotar a primeira alternativa - logo que possível, para quaisquer 2 *peers* que estejam conectados (por uma subscrição), é estabelecida a conexão TCP e o *socket* fica permanentemente aberto em tempo de execução, até que a subscrição seja removida - visto que a probabilidade de contacto entre eles é bastante elevada num sistema deste cariz, não há nenhuma desvantagem considerável em deixar os *sockets* perpetuamente à escuta.

3.2.4 Política de descarte de mensagens

Um dos requisitos do enunciado do projeto consistia em garantir a efemeridade da informação publicada pelos nodos, assegurando a sua persistência e circulação apenas durante um dado período de tempo. Desta forma, o grupo definiu um *timeout* de valor fixo (1 minuto) e programou uma rotina responsável por descartar as mensagens da base de dados que cumprissem os requisitos - terem sido originadas há um período de tempo superior ao *timeout* e terem sido já visualizadas pelo respetivo utilizador.

Esta implementação garante a efemeridade das mensagens e, ao mesmo tempo, assegura que os utilizadores têm oportunidade de as visualizar antes de serem descartadas - a limpeza indiscriminada de mensagens, com base apenas em fatores temporais, facilmente poderia originar situações em que o utilizador não se apercebia de mensagens publicadas pelos seus *peers* subscritos. A rotina referida é despoletada sempre que o nodo visualiza a sua *timeline*, visto esta ser a única operação que necessita de aceder à base de dados.

4 Conclusão

O desenvolvimento deste projeto permitiu a consolidação da matéria teórico-prática lecionada nas aulas da unidade curricular, nomeadamente da utilidade de ferramentas como o Kademlia para criar redes *peer-to-peer*, e o quão robusto tornam o sistema relativamente à tolerância a falhas e aos impactos positivos na escalabilidade.

O grupo está bastante satisfeito com o resultado final e considera que não só cumpriu os requisitos propostos no enunciado, como foi para além disso e desenvolveu uma solução escalável e dinâmica, uma vez que os módulos são independentes e modificáveis para uma outra implementação, assim como utilizou ferramentas novas e que são apropriadas ao problema em questão.

Concluindo, foi possível obter conhecimentos valiosos através da elaboração deste trabalho prático relativamente à programação por eventos, redes P2P e tabelas de *hash* distribuídas, assim como conhecimentos transversais, nomeadamente trabalho em grupo e fomento da pesquisa para obter uma solução robusta.