Introdução

O presente relatório tem como objetivo apresentar todas as decisões tomadas na conceção e implementação de um serviço de *timeline* distribuído, onde cada utilizador é também um nodo de uma rede *peer-to-peer*. Aqui, os utilizadores têm uma identidade e a possibilidade de publicar pequenas mensagens, criando a sua própria *timeline* local, e de subscrever às *timelines* de outros utilizadores, passando, assim, a receber as suas mensagens. O desenho do sistema desenvolvido prendeu-se em alguns problemas fundamentais que tinham que ser tidos em conta. Foi necessário permitir a ligação de novos nodos sem conhecimento anterior da rede e definir o protocolo de difusão das mensagens sem sobrecarregá-los, ordenando sempre as mensagens cronologicamente e evitando a sobrecarga dos nodos no armazenamento das mesmas.

Arquitetura

As vantagens dos serviços descentralizados estão no facto de não conseguirem ser encerrados num único ponto de rotura e de a carga das operações ser dividida pelos vários nós da rede, permitindo uma distribuição das tarefas de forma uniforme entre os nodos.

No início da conceção da nossa solução, tentamos responder às várias questões levantadas pelos problemas fundamentais da criação de um sistema deste tipo, moldando a arquitetura final do projeto em função delas.

De forma a combater estas problemáticas, dentro dos serviços descentralizados tivemos de optar entre sistemas de *super-peers* ou de *distributed hash table* (DHT). Sendo que uma arquitetura *super-peers* é assente essencialmente em *flooding*, optamos por usar DHT, que nos dão operações de pesquisa mais controladas, fornecendo maneiras de mapear chaves para nodos da rede e lidando também com entradas e saídas de nodos. Dentro das DHT, existem duas opções viáveis: Chord e Kademlia. Entre estas, acabamos por escolher o Kademlia, pois, ao contrário do Chord, o *routing* é simétrico, consegue baixa latência e *routing* paralelo e, além disso, existem bibliotecas em Python bem documentadas com implementações do mesmo.

Através do Kademlia, foi possível criar uma rede uniforme e descentralizada, eficiente, com tolerância a falhas em nodos da rede e guardando, de forma descentralizada, informações relativas a cada um dos utilizadores da mesma. Devido à utilização deste sistema, tivemos de optar pelo uso de ambos os protocolos de comunicação, UDP e TCP, sendo que o primeiro é usado pela implementação do próprio Kademlia para a gestão da DHT e o segundo para a comunicação desenvolvida para a troca de mensagens entre nodos, como mostra a figura.

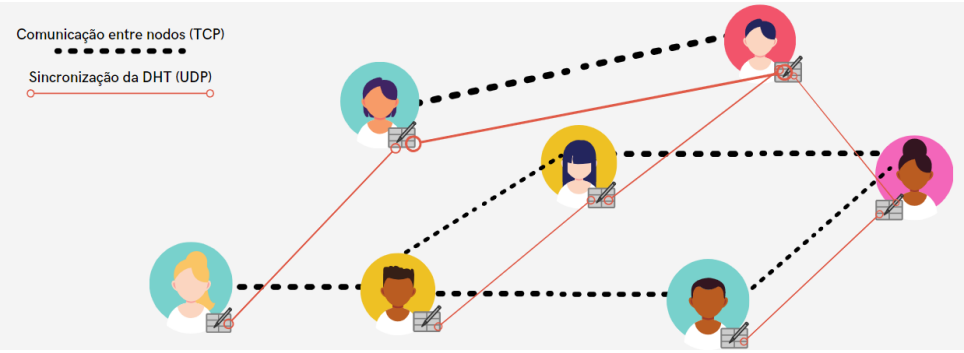

Figura 1 - Arquitetura de Rede

Outra decisão fundamental no desenho do projeto, passou pela forma de dividir a informação de cada nodo, tanto na DHT, como localmente. Decidimos, então, guardar na DHT unicamente as informações referentes a cada utilizador necessárias para o funcionamento correto da rede. Em cada entrada <*Key*,*Value*> da DHT constam as seguintes informações:

- *Key*: *nickname*, sendo um identificador único que é indicado quando o *peer* entra na rede;
- *Value*: IP do *socket* TCP, porta do *socket* TCP, lista de seguidores e o número de mensagens que já publicou.

Decidimos esta estrutura para cada entrada na DHT, visto que, com o *nickname* como *Key,* a procura por um utilizador torna-se bastante fácil e, quando este existe, tem acesso ao IP e porta TCP, caso queira enviar uma mensagem, a lista de *peers* que o estão a seguir (IP e porta), necessária para a propagação de mensagens, e o número de mensagens que este enviou, de modo a possibilitar a ordenação de mensagens.

Por outro lado, localmente, guardamos em cada nodo as informações relativas à *timeline*, às suas mensagens (*myMessages*), à lista dos utilizadores que segue (*following*) e o número de mensagens que o utilizador já enviou. A *timeline* corresponde ao conjunto das mensagens do próprio utilizador e das dos utilizadores que segue, ordenadas. A lista *following* é uma lista com informação sobre os utilizadores que este segue, tendo em cada elemento associado ao *nickname,* o número da última mensagem recebida por este. Esta lista é necessária para a ordenação de mensagens, como é explicado na secção seguinte, e para tornar possível mostrar de quem é que o utilizador recebe cada mensagem. A estrutura *myMessages* contém apenas as publicações enviadas pelo próprio utilizador e é utilizada em caso de perda de mensagens, no caso de um *peer* necessitar de repetições, possibilitando o reenvio de mensagens específicas.

A decisão de dividir a informação do utilizador entre armazenamento local e na DHT, tem como motivação não sobrecarregar a rede com informação não essencial ao funcionamento da mesma. Existe alguma informação duplicada como, por exemplo, o *nickname*, IP e porta do canal TCP, mas preferimos presumir que a mesma se mantém estática e, deste modo, não sobrecarregar a rede. A implementação de persistência dos dados, quando o nodo é desligado era importante, mas decidimos focar esta abordagem principalmente na difusão de mensagens e na efemeridade das mesmas, de modo a tonar o sistema escalável para *peers* que tenham muitos seguidores e que recebam um elevado número de mensagens.

Implementação

Tal como referido anteriormente, o nosso serviço foi contruído sobre bibliotecas consolidadas da linguagem Python como a Kademlia e *asyncio*. A biblioteca Kademlia, tal como o nome indica, é uma implementação assíncrona da DHT Kademlia que usa a biblioteca *asyncio* para garantir comunicação assíncrona, sendo que os nodos comunicam sobre ligações UDP.

Ligação de um nodo à rede

A primeira questão fundamental ao sistema que surgiu na sua implementação foi relativa à conexão de um novo nodo na rede P2P com a ausência de um servidor centralizado. Optamos, então, por compor a nossa rede com dois tipos de nodos: nodos regulares e nodos *bootstrap*. Ao iniciarmos um novo nodo, este recebe como parâmetros uma porta TCP, uma porta UDP e a porta do nodo *bootstrap*. Caso seja o primeiro nodo, a porta do nodo *bootstrap* deve ser 0.

Para um novo nodo se ligar à rede, comunica com um nodo *bootstrap*, sendo, assim, introduzido na DHT e passando a estar conectado à rede *peer-to-peer*. O que diferencia estes dois nodos é a porta do nodo *bootstrap* que este recebe como *input*. Sempre que esta porta se encontra a 0, significa que o próprio nodo é um nodo *bootstrap*, caso contrário, liga-se ao nodo *bootstrap* da porta que recebeu.

Seguir novos utilizadores

Quando um nodo pretende seguir um novo utilizador, começa por aceder à DHT e verificar se o utilizador de facto existe, fazendo um *get* do seu *nickname*. Caso exista, verifica se já se encontra a segui-lo e, em caso negativo, é adicionado um tuplo com o IP e porta à lista de seguidores do utilizador, atualizando a lista na DHT.

Difusão de mensagens

Um utilizador, sempre que publica uma mensagem, apenas consulta o valor do seu próprio *nickname* na DHT e sabe automaticamente quem é que o está a seguir e, por isso, a quem necessita de se conectar. Alguns metadados são acrescentados à mensagem como, por exemplo, o número de publicações do utilizador incrementado, de modo que quem receba possa verificar a ordenação da mesma e, também, o tipo da mensagem, que indica se esta precisa de ser retransmitida ou não, e, no caso de ser, para quem.

De forma a não sobrecarregar os nodos que têm um elevado número de seguidores no momento de difusão das suas mensagens e de tornar o sistema escalável nessa matéria, desenvolvemos um algoritmo de difusão que pré-define um limite X máximo de cópias de uma mesma mensagem que cada nodo pode enviar e, caso um nodo tenha que enviar um número de mensagens superior a esse limite X, a mensagem é enviada para X nodos que ficam responsáveis de distribuir a mensagem pelos restantes nodos destino, repetindo o algoritmo sucessivamente, garantindo que chega a todos. É importante ainda referir que o limite X não é ultrapassado nos nodos recetores da mensagem, ou seja, se um nodo ficar responsável por reencaminhar uma mensagem a um número de nodos superior ao limite X previamente definido, reencaminha a mensagem apenas para X desses nodos e estes ficam responsáveis por redistribuir para os restantes. Ou seja, a mensagem faz os saltos necessários de forma a respeitar sempre o limite de mensagens que cada nodo pode enviar.

É de notar que as conexões no nosso sistema são abertas e fechadas no momento da propagação da mensagem, pois, embora isto não seja o ideal, visto que a cada mensagem abre uma nova conexão, manter as conexões abertas também acarreta os seus custos. Deste modo, decidimos focar na difusão controlada das mensagens com um número máximo de conexões,

como foi explicado nos pontos anteriores, pois consideramos um ponto fulcral a escalabilidade da nossa aplicação, em detrimento de alguma performance perdida pela criação das conexões.

Ordenação das mensagens

Inicialmente, ponderamos o uso de *vector clocks* para a ordenação das mensagens dos *peers*, mantendo o registo do mesmo na DHT. Esta decisão traria alguns problemas, visto que o *vector clock* vai crescer consoante o número de utilizadores que o *peer* seguir e, deste modo, ser um entrave para a escalabilidade do sistema. Sendo assim, optamos por apenas guardar um contador do número de mensagens publicadas por esse *peer* na DHT. Desta forma, quando o utilizador segue um *peer*, guarda quantas mensagens esse *peer* já publicou e, quando começar a receber mensagens do mesmo, pode comparar o valor que tem armazenado localmente, na lista de *following*, com o da mensagem, e, se for o valor seguinte, aceita a mensagem. Caso contrário, pede repetição desde a última mensagem que conhece até à mensagem que recebeu.

Todas as mensagens enviadas no sistema contêm, também, um ID da *timestamp* do momento da publicação que, quando chega ao utilizador, é traduzido para o tempo local via servidor NTP, permitindo, assim, associar e ordenar os tempos físicos de cada mensagem, independentemente da *timezone* em que se encontra cada nodo da rede. A implementação de tempos físicos foi, na nossa opinião, importante, porque, embora adicione uma complexidade adicional, por se tratar de uma timeline, achamos fulcral a existência de tempos físicos das mensagens.

Efemeridade das Mensagens

De forma a evitar a sobrecarga dos nodos no armazenamento das mensagens, implementamos uma lógica de efemeridade das mensagens semelhante à funcionalidade das *stories* que existe em redes sociais populares, como é o caso do Facebook ou do Instagram. Achamos que a implementação da efemeridade de mensagens era extremamente importante para garantir uma maior escalabilidade, principalmente em casos de utilizadores que sigam muitos outros utilizadores, tendo consciência que, obviamente, o utilizador perde alguma informação na sua *timeline*.

Neste caso concreto, a ideia do algoritmo passa por manter as mensagens guardadas até receber uma nova mensagem com uma diferença temporal superior a 24 horas, ou seja, sempre que é recebida uma mensagem, um nodo verifica se ainda guarda alguma mensagem com mais de 24h e, em caso afirmativo, descarta-as.

Métricas

Conhecido pela sua eficiência, o Kademlia garante que uma procura na árvore tem um máximo de O(log n) saltos.

Relativamente ao algoritmo de difusão de mensagens, para um exemplo com número do limite máximo de 20 conexões, com 20 ou menos seguidores, há apenas 1 salto. Entre 20 e 420 mensagens a difundir, 2 saltos e, da mesma forma, para um número de conexões entre 420 e

8420, temos apenas 3 saltos, e assim sucessivamente. A tabela abaixo representa, para cada número de saltos máximo necessário, quantos seguidores conseguirão ser atingidos.

Nº de Saltos	Nº de Nodos
1	20
2	420
3	8 420
4	168 420
5	3 368 440

Figura 2 - Métrica do número de saltos do algoritmo de difusão das mensagens

Conclusões

Concluindo, o serviço desenvolvido tem em consideração os aspetos que julgávamos ser mais importantes ao sistema, tais como a difusão, efemeridade e ordenação causal das mensagens. Pensamos ter atingido com sucesso os objetivos propostos e fazemos um balanço positivo do resultado final do sistema obtido, pois garantimos uma resposta sólida às questões fundamentais do projeto nos pontos descritos acima. Quanto à ligação de novos nodos sem conhecimento anterior da rede, a abordagem com nodos regulares e nodos *bootstrap*, revelou-se eficiente e funcional. Podemos, também, verificar que o protocolo de difusão das mensagens sem sobrecarregar os nodos é extremamente escalável e permite difundir mensagens por milhões de seguidores em poucos saltos na rede, tal como explicitado no exemplo acima. Quanto à ordenação cronológica das mensagens, tanto através da utilização de IDs ordenados específicos para as mensagens de cada utilizador, como da associação de um tempo físico do momento de publicação independente da *timezone* do nodo recetor, permite que as *timelines* apresentem *timestamps* corretas e de forma ordenada e sejam sensíveis à perda de mensagens na rede, garantindo, também, causalidade. Por último, de modo a evitar a sobrecarga dos nodos no armazenamento de mensagens, consideramos ter adotado a melhor estratégia, implementando efemeridade das mesmas com base num prazo temporal lógico e verificado apenas na receção de uma mensagem, evitando a constante execução desse cálculo.

Embora estejamos bastante satisfeitos com o resultado apresentado, sabemos que ainda existem alguns pontos que poderiam ser abordados, de forma a ter uma aplicação completa e robusta o suficiente para um contexto real, embora pensemos que não fosse esse o objetivo deste projeto. Em futuras revisões do nosso sistema, seria possível fazer melhorias em certas áreas de forma a criar um sistema mais robusto e aplicável a uma situação de utilização real, essencialmente do ponto de vista do utilizador.

Uma das mais importantes melhorias que poderiam ser feitas no sistema, seria permitir aos seguidores reencaminharem mensagens de utilizadores que seguem e que estejam offline a outros nodos. Na difusão de mensagens, poderíamos introduzir um limite de cópias a reencaminhar como um valor dinâmico, tendo em conta, assim, quantos pedidos de difusão de mensagens diferentes existem, para definir um limite de cópias de cada mensagem, de modo a não sobrecarregar os nodos e tornando o sistema ainda mais escalável. Por fim, poderíamos introduzir consistência no nosso sistema, permitindo aos utilizadores entrar e sair da rede sem perder os seus dados de sessão. Com esta introdução, talvez fizesse sentido introduzir, também,

um sistema de autenticação, que levantaria ainda algumas questões de difícil análise, caso quiséssemos tolerar falhas bizantinas, por exemplo.

Num contexto mais pessoal, achamos, desde o início, o projeto bastante desafiante e ajudou-nos a explorar e compreender as nuances de redes *peer-to-peer* e aprofundar outros conhecimentos também lecionados ao longo da Unidade Curricular de Sistemas Distribuídos em Larga Escala, num ponto de vista mais prático.

Referências

Kademlia *Documentation*, https://kademlia.readthedocs.io/en/latest/
asyncio *Documentation* , https://docs.python.org/3/library/asyncio.html
Repositório NTP *servers*, https://github.com/Tipoca/ntplib