# Timeline Descentralizada

Cecília Soares A34900, Diogo Ribeiro A84442, Joel Ferreira A89982, Luís Cunha A84244, and Rui Mendes A83712

Sistemas Distribuídos em Larga Escala
Universidade do Minho

**Resumo** O presente relatório descreve a arquitetura do nosso serviço de *Timeline* distribuído, usando uma rede *peer-to-peer*, bem como o seu funcionamento e ainda uma análise sobre os resultados obtidos.

## 1 Introdução

Neste relatório iremos detalhar a conceção e implementação de uma *Timeline* Distribuída. Este serviço consiste numa rede *peer-to-peer* em que cada utilizador possui um *username* e pode efetuar diversas ações, nomeadamente publicar pequenas mensagens textuais e subscrever outros utilizadores para ter acesso às mensagens que os mesmos publicam. Um conjunto de publicações, de um ou mais utilizadores, é referido como uma *timeline*.

Um dos objetivos principais é permitir que utilizadores que subscrevem outros utilizadores possam armazenar as suas *timelines* localmente durante um pequeno período de tempo, de modo a ajudar na difusão das mesmas. O conteúdo remoto encontra-se disponível quando o utilizador que publica, ou algum dos seus subscritores, se encontram *online*.

## 2 Implementação

Antes de iniciarmos o processo de implementação, debatemos algumas opções e trocámos ideias acerca de diferentes abordagens que poderíamos seguir. Após esta discussão chegamos à conclusão de qual seria a arquitetura indicada para o problema, sendo que a mesma será analisada neste capítulo. Uma das primeiras decisões por nós tomadas consistiu na utilização da linguagem **Python** para a implementação do projeto, dado o ecossistema de ferramentas existentes e a nossa experiência prévia.

### 2.1 Arquitetura

Dado que o objetivo é ter uma rede *peer-to-peer*, e considerando o problema em questão, considerámos que a utilização do **Kademlia**[1] iria ser essencial para a execução do projeto. O Kademlia consiste numa ***Distributed Hash Table***, doravante (DHT), que considerámos ser uma ferramenta adequada e

bastante útil ao desenvolvimento do projecto, dado que consiste numa *hash table* distribuída desenhada para redes de computadores *peer-tp-peer* descentralizadas.

Acresce que, o Kademlia tem melhor performance que outros algoritmos, como, por exemplo, o *Chord*[2]. Algumas das vantagens que nos fizeram optar por este algoritmo passam pelo *routing* ser **simétrico** e pelo número máximo de contactos no processo de procura ser da grandeza ***O(log(n))***.

O nosso objetivo ao utilizar uma DHT passa por sermos capazes de associar a um determinado utilizador um valor ou conjunto de valores. Além de nos permitir fazer essa associação, a utilização destas estruturas permite-nos que todos os utilizadores a possam consultar, mesmo perante a **falha ou ausência de outros utilizadores da rede**.

A estrutura de uma entrada na DHT consiste numa **chave**, que, no nosso caso, é o ***username***, dado que é um valor único, e num **valor**, que consiste num **objeto JSON** convertido para string(porque a DHT não suporta objetos JSON). Este objeto é constituído pelos seguintes campos: **tcp_port**, que é a **porta tcp do utilizador** mapeado; **followers**, que é um *array* que contém **os seguidores do utilizador** mapeado; e **vector_clock** que consiste num **relógio lógico**. O array *followers* contém todos os seguidores do utilizador mapeado, tendo associado a cada utilizador, a sua **porta tcp**. Optámos por utilizar apenas a porta tcp dado que todos os utilizadores estão na mesma rede(e máquina). No entanto, num ambiente de produção, esta solução iria necessitar de uma revisão. O campo *vector_clock*(relógio lógico) é também um array que contém os *usernames* presentes no *array followers* associados a um **contador**. Este contador representa o **identificador da mensagem mais recente** que o utilizador possui do utilizador que é seguido. Além dos *followers*, a primeira entrada no relógio lógico é a do utilizador mapeado, sendo que este terá sempre o contador atualizado.

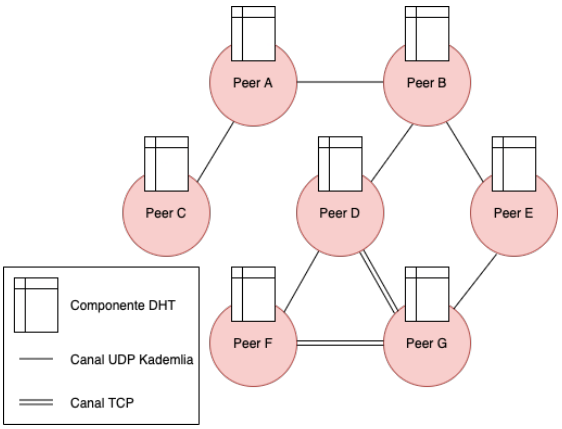


**Figura 1.** Exemplo com 7 *peers* onde o *Peer* G contacta diretamente via TCP os *Peers* D e F.

Paralelamente à utilização de uma DHT, os utilizadores poderão comunicar diretamente uns com os outros através de **canais TCP**, usando as portas anteriormente mencionadas. A utilização deste tipo de canais prende-se, principalmente, com o facto de pretendermos ter canais fiáveis e ordem na entrega de mensagens, (algo que canais UDP não garantem). Os utilizador comunicam através dos canais TCP para realizar operações como: seguir utilizadores, publicar mensagens ou pedir *timelines*. Para o envio destas operações decidimos que o uso de **Protocol Buffers** seria indicado dado que nos permitem definir esquemas representativos das diferentes operações.

Relativamente ao armazenamento das *timelines*, utilizamos uma biblioteca em *Python* para lidar com ficheiros *YAML*. Optámos por este formato dado que é inteligível e fácil de compor. Os dados de um utilizador armazenados em ficheiro consistem na *timeline* do utilizador.

Por último, em memória temos uma estrutura de dados que mantém a lista de utilizadores que um dado utilizador segue, a qual deve conter em cada uma das suas posições, a identificação do utilizador seguido e as últimas 10 mensagens que o utilizador recebeu deste, sendo que cada mensagem tem um número de sequência e um *timestamp*.

| **Campo** | **Conteúdo** |
|---|---|
| ***tcp_port*** | Porta TCP do utilizador mapeado. |
| ***followers*** | Array que contém os usernamse dos seguidores e a respetiva porta TCP. |
| ***vector_clock*** | Array que contém os usernames do utilizador mapeado e dos seus seguidores, associados a um contador. |

**Tabela 1.** Campos da informação associada a um *username* nda DHT.

### 2.2 Operações

De seguida iremos listar as principais operações que o sistema permite que um utilizador execute.

A operação de **Seguir Utilizador** permite que um determinado utilizador possa seguir outros utilizadores da rede. Nesta operação, quando um utilizador fornece um *username* para seguir, além de verificar se o utilizador já o segue, verificamos se se trata do próprio utilizador. Após estas verificações, o primeiro passo consiste em aceder à DHT, procurando a entrada referente ao *username* fornecido. O passo seguinte consiste em aceder ao objeto presente no campo *value*, e inserir o par *username* e porta tcp no campo *followers*. Além disso, o *username* é inserido no campo *vector_clock*, sendo o seu contador igual a 0 (dado que ainda não possui mensagens). Após estas alterações, o objeto é inserido na DHT, atualizando a informação do utilizador seguido. Por fim para o utilizador se atualizar localmente com a timeline desse utilizador que acabou de

seguir é pedida a sua timeline, caso o utilizador seguido tenha publicado alguma mensagem.

A operação **Publicar Mensagem** permite que um utilizador possa publicar uma pequena mensagem de texto que será disponibilizada na sua *timeline*, sendo esta disseminada pelos restantes utilizadores que o seguem e que estiver *online*. A transmissão das mensagens publicadas é feita da seguinte forma: o utilizador verifica o vector clock de 20% dos seus *followers* para saber qual a última mensagem recebida por estes, para lhe enviar as mensagens em falta até à presente publicação. Depois de recebidas estas mensagens, os seus seguidores actualizam as suas respectivas posições no vector clock relativo ao utilizador seguido e consultam o seu vector clock para propagarem as publicações do *publisher*, a N/2 seguidores que ainda não as tenham, entretanto, atualizadas. O processo repete-se até que todos os followers do utilizador que estejam ligados tenham a sua entrada no vector clock atualizada. Em cada utilizador que receba esta mensagem de *broadcast* é obtida a entrada do utilizador que enviou inicialmente a mensagem de broadcast e verificado se o *vector clock* dos sesus *followers* já observaram a mensagem que está a ser *broadcasted*, caso seja verdade então terminamos o processo de *broadcast*.

A operação **Pedir *Timelines*** permite que um utilizador verifique se possui as *timelines* de todos os utilizadores que segue atualizadas, e caso não possua, efetua pedidos das mensagens em falta. Acresce que, se um seguidor estiver desligado aquando de uma nova publicação do utilizador, este não irá receber a nova mensagem publicada. Neste caso, quando se juntar novamente à rede, este deve consultar a sua lista local de pessoas que segue, verificar na DHT os vector clocks de cada uma delas e o seu, caso haja utilizadores que este segue que tenham mais mensagens do que aquelas que este já recebeu, é seleccionado aleatoriamente um utilizador *online*, entre aqueles que tenham a *Timeline* deste mais atualizada, para receber o pedido de mensagens em falta do utilizador que agora se juntou à rede.

### 2.3 Melhorias introduzidas

Após implementarmos uma versão inicial do sistema, consideramos que existiam aspectos arquitecturais que podiam ser melhorados com vista a aumentar a performance e o desempenho do nosso programa. Em primeiro lugar, a lista de *followers* de determinado utilizador deve ser também armazenada localmente, o que iria permitir um acesso mais rápido à mesma para realizar a disseminação das mensagem pelos seguidores do utilizador, evitando-se, assim, um acesso à DHT. Não ignoramos que a referida lista armazenada localmente pode estar ligeiramente desatualizada, todavia, como a transmissão das mensagens é um esforço repartido entre o *publisher* e os seus seguidores, basta que a mencionada lista não tenha uma enorme discrepância com a da DHT para que se ganhe em termos de desempenho. Convém notar que esta lista local seria regularmente atualizada para evitar divergências significativas.

Em segundo lugar, para evitar que sempre que um utilizador inicia uma sessão tenha de ir à DHT consultar todos os relógios lógicos dos utilizadores que

segue e verificar se estão de acordo com as mensagens que este recebeu, pensamos em guardar localmente a porta tcp de cada utilizador para evitar as consultas à DHT. Com efeito, apenas consultaríamos a DHT caso o utilizador esteja *offline*. O problema associado a esta alternativa está relacionado com utilizadores que tenham inúmeros seguidores, pois tem elevada probabilidade de estarem sempre a entrar e a sair do sistema seus seguidores, o que pode congestionar a rede e comprometer o desempenho.

## 3 Análise de Desempenho

Após terminarmos a implementação do sistema, consideramos que seria útil tentar perceber quais as limitações do mesmo em termos de performance. Com esse objetivo em mente, recorremos a *benchmarks* para medir a eficiência do sistema.

Assim, consideramos que seria útil obter as seguintes métricas: o ***throughput*** da rede em operações por segundo, e o **tempo de resposta** das operações em milisegundos. Decidimos também que seria interessante para a nossa metodologia, correr cada *benchmark* **5** vezes, de modo a obter resultados com menor margem de erro. Para os cenários de teste em causa, decidimos que a carga de trabalho mais relevante consistiria em todos os utilizadores publicarem mensagens(100 mensagens), sendo que todos os utilizadores seguem um outro utilizador. Deste modo conseguimos, de uma forma simples, submeter a rede a alguma carga. Não considerámos a hipótese de todos os utilizadores seguirem todos os utilizadores dado que iria gerar uma carga pouco realista.

Dadas as limitações de *hardware*, e dado que todos os nodos da rede se encontram na mesma máquina, considerámos quer para todos os cenários de teste iria existir **1** ***bootstrap node***. Relativamente a utilizadores, considerámos cenários para **2**, **3**, **5** e **10** utilizadores da rede.

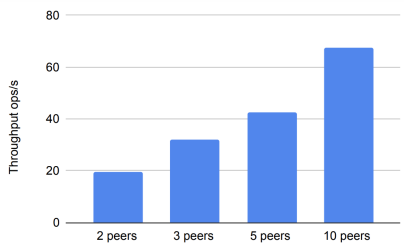


**Figura 2.** Resultados da medição do *Throughput*

Analisando os resultados do *Throughput*(Figura 2) podemos inferir que esta métrica aumenta de uma forma relativamente linear. No caso dos Tempos de Resposta(Figura 3), notámos que estes se assemelham mais a uma subida expo-

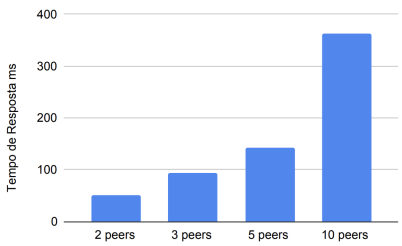


**Figura 3.** Resultados da medição do Tempo de Resposta

nencial. No entanto, para obter dados mais fidedignos seria importante aumentar o número de nodos da rede, bem como o número de execuções.

## 4 Conclusão

O nosso serviço foi desenvolvido tendo em conta os aspectos que consideramos relevantes, nomeadamente a distribuição da carga de trabalho de propagação das mensagens por vários nodos, a escolha aleatória de nodos que irão transmitir as mensagens, de modo a não sobrecarregar os nodos que têm mais seguidores.

Como trabalho futuro, seria interessante que o nosso programa permitisse que os utilizadores acedessem ao sistema através de diferentes dispositivos, com diferentes endereços. Acresce que, o nosso serviço de *social networking* seria mais realista tendo a possibilidade de efetuar a operação de *unsubscribe* de determinado utilizador anteriormente seguido.

Apesar de considerarmos que tivemos sucesso na nossa implementação, chegamos à conclusão que algumas questões relevantes ficaram por resolver e seriam de extrema importância numa iteração futura, sendo a questão mais importante o controlo de concorrência nas entradas da DHT. A falta de controlo de concorrência na DHT permite que diferentes utilizadores possam efetuar leituras e escritas simultaneamente, podendo causar inconsistência nos dados.

## Referências

1. Kademlia Documentation, https://kademlia.readthedocs.io/en/latest/. Acedido a 20/05/2021.
2. Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications, https://pdos.csail.mit.edu/papers/ton:chord/paper-ton.pdf. Acedido a 21/05/2021.
3. Kademlia: A Peer-to-peer Information System Based on the XOR Metric, https://pdos.csail.mit.edu/ petar/papers/maymounkov-kademlia-lncs.pdf. Acedido a 21/05/2021.