Sistemas Distribuídos em Larga Escala

Decentralized Timeline

Daniel Fernandes (a78377) Maria Helena Poleri (a78633)
Mariana Miranda (a77782)

17 de Maio de 2019

Resumo

O presente documento tem a intenção de apresentar a solução proposta para o desenvolvimento de um serviço *peer-to-peer* de *timeline* descentralizado, no qual cada utilizador pode publicar na sua própria *timeline*, assim como subscrever à *timeline* de outros utilizadores, podendo ainda guardar e auxiliar na propagação do conteúdo daqueles a que subscreveu.

1 Introdução

O presente documento serve o propósito de descrever a criação de um serviço de *timeline* descentralizado, onde um utilizador representa um nodo numa rede *peer-to-peer*. Cada *peer* pode publicar mensagens formando na sua máquina uma *timeline* local, bem como subscrever à *timeline* de outros utilizadores, ajudando adicionalmente a armazenar e redirecionar o conteúdo desses. Um utilizador tem acesso às mensagens mais recentes de quem se encontra subscrito, quando esse ou um dos seus subscritores, se encontra *online*. Este serviço tem ainda o cuidado de preservar uma mensagem apenas por um certo período de tempo, caso contrário a quantidade de informação cresceria exponencialmente.

A vantagem de escolher um sistema descentralizado, reside em grande parte no facto de este não conseguir ser encerrado, uma vez que não existe um só ponto de ataque, assim como na repartição de carga pelos vários nodos da rede. Para além disso, tem a vantagem de que nenhuma publicação pode ser censurada: como não existem servidores, não é possível eliminar publicações de um utilizador.

Adicionalmente, este serviço garante ainda ordenação causal das mensagens trocadas entre *peers*, assim como providencia informação acerca do tempo físico em que estas foram enviadas. Foi ainda garantido que, caso um *peer* fique *offline*, uma vez que volte a ficar *online* tem acesso às mensagens que perdeu durante esse período.

2 Trabalho relacionado

A implementação de sistemas *peer-to-peer* remonta ao início da Internet, tendo-se tornado especialmente popular em sistemas de partilha de ficheiros. Esta arquitetura distribuiu o custo de partilha de dados pelos vários nodos da rede, funcionando estes quer como cliente, quer como servidor, contrariamente ao que acontece no tradicional paradigma cliente-servidor, permitindo assim o escalonamento de aplicações sem a necessidade de servidores avançados e caros [1].

Para além de se poder verificar a influência de *designs* P2P em sistemas primitivos como Usernet News, DNS, Bayou, entre outros, é possível dividir a evolução destes sistemas em 3 gerações [2]. Na primeira geração a procura era centralizada, sendo exemplo disso o sistema de partilha de música Napster. A segunda geração procurou resolver problemas de escalabilidade, anonimato e tolerância a faltas, propondo uma arquitetura descentralizada e não estruturada, podendo se verificar em sistemas como Gnutella e Kazza. Foi apenas na terceira geração que surgiram camadas de *middleware*, estabelecendo um sistema descentralizado e estruturado que permite operações de atribuição e procura de um modo mais organizado e eficiente. Os sistemas mais conhecidos desta geração são o Pastry, Tapestry, CAN, Chord e Kademlia.

3 Arquitetura

Durante a fase de decisão de qual seria a arquitetura a escolher, decidimos pelo uso de uma totalmente descentralizada, fazendo uso do Kademlia [3] para a sua descrição. Com esta abordagem, sabíamos que o processo de procura necessitava de no máximo $O(\log n)$ contactos, o que é extremamente eficiente. Consideramos também o uso de *super-peers* mas essa não era uma solução totalmente descentralizada e portanto mais sujeita a ataques DoS, e não sobrecarregava nenhuns nodos em particular (com a abordagem de *super-peers* todas as mensagens da rede passam por estes). A maneira como estruturamos a solução encontra-se descrita nas secções seguintes.

3.1 Arquitetura de Rede

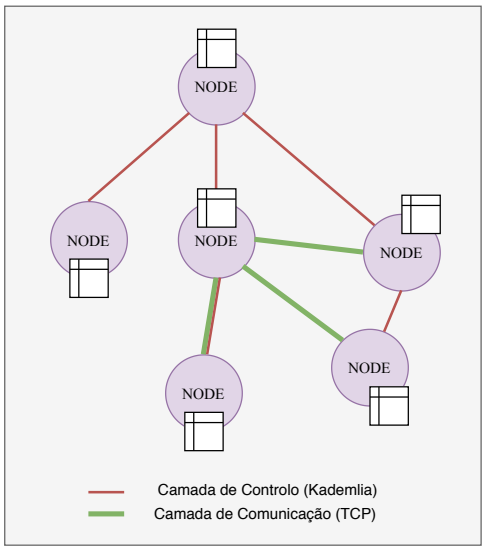

Figura 1: Camadas de rede

Em termos de arquitetura de rede, decidimos que os nodos iriam comunicar entre si usando tanto conexões UDP, como conexões TCP. As primeiras referem-se às conexões geridas pelo próprio Kademlia, que garantem que as informações da DHT sejam distribuídas pelos vários nodos. As conexões TCP, por sua vez, correspondem às ligações que permitem aos nodos trocar os *posts* que publicam entre si. Estas últimas são criadas sempre que um utilizador segue outro, como será mais a fundo descrito na secção que aborda a difusão de mensagens (Secção 4.1).

3.2 Estrutura da entrada da DHT

Uma das decisões que tivemos de fazer na fase inicial do trabalho, relacionava-se com decidir qual seria o conteúdo de cada entrada na DHT. Assim, decidimos que a chave da *hashtable* seria o nome de utilizador do *peer* e as informações que necessitaríamos de guardar acerca dele, o seu valor, as que se encontram na Tabela 1. O endereço e porta de escuta de um nodo são essenciais para estabelecer conexões entre nodos. A necessidade dos restantes campos será explicada ao longo da Secção 4.

ip	O endereço do *peer*
port	Porta para escutar conexões TCP
followers	Lista de *usernames* que seguem utilizador
following	Lista de *usernames* que o utilizador segue
redirect	Mapa de redirecionamentos pelos quais o utilizador está responsável
msg_nr	Número da última mensagem enviada pelo utilizador

Tabela 1: Campos do valor de uma entrada (um utilizador) na DHT

4 Implementação

A implementação do sistema foi realizado na linguagem Python que, apesar de não nunca ter sido largamente utilizada pelo grupo, foi adotada nos restantes trabalhos da cadeira tendo proporcionando boas impressões. O projeto recorreu ainda a dois módulos da linguagem que foram imprescindíveis para o desenvolvimento do sistema, Asyncio e Kademlia. O Kademlia tal como o nome indica diz respeito a uma implementação de uma tabela de *hash* distribuída, tendo sido útil como componente "central" de informação da rede. Já o Asyncio é um módulo que materializa o paradigma de programação por eventos em Python. Sendo o sistema em questão essencialmente "IO Bound" consideramos que era o paradigma que melhor se adaptava às nossas necessidades, evitando a criação de múltiplas *threads* e prevenindo problemas de controlo de concorrência.

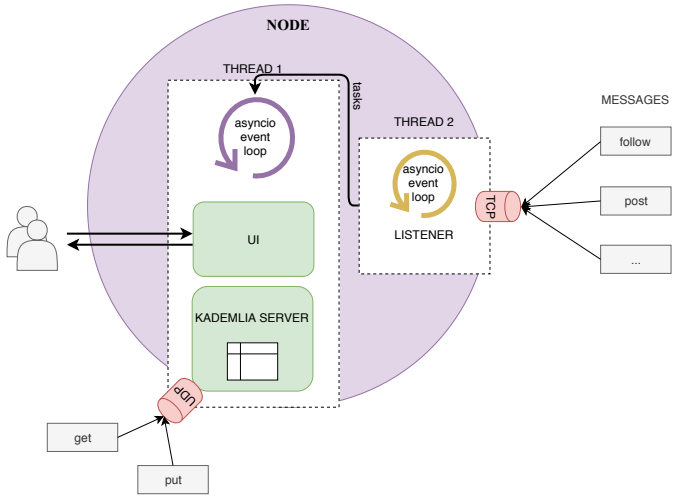

Figura 2: Funcionamento de um *peer*

4.1 Difusão de mensagens e hierarquias

Grande parte da funcionalidade do sistema depende de um aspeto fundamental que é a difusão de mensagens. Neste sistema em concreto é necessário que as mensagens das publicações fluam do nodo origem para os seus subscritores, sendo que, naturalmente, outras mensagens serão igualmente necessárias para o correto funcionamento da aplicação (mensagens de controlo, etc). Focando-nos então nestas mensagens, a decisão principal consistia na forma em como organizar as ligações entre os nodos, isto é, o estabelecimento de conexões.

Numa primeira versão era criada uma conexão para cada um dos seguidores sempre que se pretendia enviar uma mensagem. Contudo, sendo o estabelecimento de conexões um processo pesado, esta solução não era de todo sensata e rapidamente foi abandonada.

Uma segunda alternativa consistiu no estabelecimento de conexões com todos os seguidores aquando do momento de autenticação, conexão esta que era mantida durante todo o período de atividade. Esta alternativa vinha colmatar o esforço de estabelecimento de conexões visto anteriormente, mas introduzia um outro problema igualmente mau que era o de possuir muitas conexões abertas em simultâneo, sendo que facilmente se poderia ultrapassar o limite de conexões suportado pelo sistema operativo.

A solução final baseou-se numa organização hierárquica. Foi definido um determinado limite de seguidores a partir do qual se escolhe um número reduzido dos mesmos para ficarem responsáveis por encaminhar as mensagens para o novo nodo. A escolha deste seguidores é realizada por cada novo seguidor e com base no número de seguidores e nodos aos quais tenham de reencaminhar que já possuam. De referir ainda que se optou por se estabelecer as conexões no momento de envio da primeira mensagem, isto porque, fazendo um paralelo com outras aplicações idênticas, várias são as vezes que se inicia a aplicação sem o objetivo de efetuar uma publicação.

4.2 Ordenação causal das mensagens

Tratando-se o caso de uso de uma *timeline*, era importante garantir ordem causal entre as mensagens enviadas por cada *peer* na rede, de forma a que nenhuma mensagem que tenha sido origem de outra apareça depois dessa. A primeira abordagem na qual pensamos para resolver tal problema foi o uso de relógios lógicos, mas essa alternativa revelou sofrer de alguns problemas. O uso de relógios lógicos neste contexto garantia a ordenação causal das mensagens, mas requeria coordenação global, algo que afetaria o sistema de tal forma que poderia deixar de ser usável. Por exemplo, caso um nodo entrasse no sistema, todos os outros teriam de ser notificados para atualizar o próprio relógio lógico que, adicionalmente, para uma quantidade muito grande de *peers*, se tornaria num peso muito grande a enviar em cada mensagem gerada.

Após uma pesquisa acerca de como se faz esta ordenação num contexto real, deparamo-nos com o algoritmo que o Twitter usa para gerar identificadores únicos para *tweets*, utilizadores etc., o Snowflake [4]. Neste são gerados identificadores constituídos por um *timestamp*, que identifica o tempo em que a mensagem foi gerada, um identificador da máquina, que serve como desempate caso duas mensagens vindas de diferentes utilizadores sejam geradas ao mesmo tempo e um número de sequência que, para o mesmo utilizador, evita que sejam gerados dois *timestamps* iguais no mesmo *clock tick*.

O problema da implementação do Twitter Snowflake para o nosso caso é o facto de este requerer coordenação para preencher o campo de identificação da máquina origem, sendo necessário adaptá-lo ao nosso caso. Com mais uma pesquisa, chegamos à implementação Flake [5] que, mediante o uso de identificadores maiores (128 bits), funciona de forma totalmente descentralizada, fazendo uso do endereço MAC para a identificação de uma máquina.

Na nossa implementação adaptamo-lo de uma forma que achamos mais conveniente, usando para identificador da máquina o endereço IP que já estávamos a usar previamente para as conexões TCP entre nodos. Em relação ao campo do *timestamp*, para evitar pedidos constantes ao NTP, fazemos periodicamente sincronização do relógio da máquina com este, fazendo assim que duas pessoas a grande distância tenham a mesma noção do tempo.

Figura 3: Formato de um *flake* na nossa implementação

Para além da ordenação causal, esta alternativa dá-nos ainda acesso ao tempo físico de uma publicação independentemente da *timezone* do nodo destino (o *timestamp* UTC é imediatamente traduzível para o tempo local), o que enriquece a informação que a nossa *timeline* pode providenciar.

De forma a ser possível um utilizador saber se faltam mensagens de um determinado utilizador (quando recebe uma mensagem mais recente do que outras que ainda não chegaram) necessitamos adicionar um número de mensagem (`msg_nr`) associado a cada utilizador, que indica em que mensagem este se encontra, incrementado a cada envio. Neste caso, não se poderiam usar os *flakes* por estes poderem ter intervalos arbitrários entre si. Assim, se um utilizador receber uma mensagem 9 de um utilizador e não tiver a 7 e a 8, sabe que não a pode ver antes de ter acesso a estas.

4.3 Arranque da aplicação e recuperação

Quando um determinado utilizador, já registado, se autentica na aplicação, após um período no qual se encontrou ausente, necessita recuperar o seu estado. Parte desse estado é obtido pela realização de um pedido GET, com o nome de utilizador respetivo, à *hashtable* distribuída através de um servidor do Kademlia instanciado no momento de arranque da aplicação. A resposta a este pedido inclui informações úteis tais como a identificação de cada um dos seus seguidores, como também das pessoas que se encontra a seguir, juntamente a estas últimas é mantido um identificador da última mensagem recebida/processada com origem nas mesmas. Ao manter esta informação em memória facilmente atualiza e submete para a rede, não necessitando de realizar o mesmo pedido novamente.

O restante estado diz respeito às mensagens propriamente ditas que constituem os dados fulcrais deste tipo de aplicação. Relativamente às mensagens que um utilizador possuía quando se desconectou da aplicação, dado que existem localmente, são facilmente importadas e recuperadas. Contudo, as publicações ocorridas no momento em que o utilizador se encontrava *offline*, com origem noutros utilizadores seguidos pelo mesmo, não se encontram na sua máquina local pelo que têm de ser obtidas. A estratégia passou por, para cada utilizador, calcular os identificadores das mensagens em falta e tentar estabelecer conexão com a origem. Caso a origem não se encontre disponível, é possível conhecer os seus restantes seguidores e caso se encontrem mais atualizados que o próprio é realizada uma tentativa de contacto e requisição das mensagens em falta, até que se atinja o conhecimento completo.

4.4 Política de descarte de mensagens

Um dos requisitos deste sistema consistia na efemeridade das mensagens, pois estas só devem ser guardadas e enviadas para os seus subscritores por um dado período de tempo, caso contrário, o número de mensagens no sistema cresceria exponencialmente, o que não é o desejado.

A fim de ir ao encontro deste critério, foram consideradas duas hipóteses. Uma primeira alternativa consistia em ter uma *thread*, de x em x segundos, a eliminar as mensagens que tinham sido criadas há um dado período de tempo, no entanto, este processo implicaria o uso de mais recursos e a que, caso um utilizador durante esse intervalo não recebesse mensagens, a sua *timeline* estivesse vazia.

Para colmatar estes problemas foi proposta uma segunda alternativa que se baseia em despoletar a verificação e eliminação de mensagens antigas, caso estas já tenham sido lidas, assim que uma nova mensagem fosse recebida. Esta solução também tem as suas desvantagens, pois poderia ser potencialmente pesado se estivesse continuamente a receber mensagens, contudo,

consideramos que os pontos positivos ultrapassam os negativos e que se traduz na melhor resposta ao problema proposto.

5 Conclusões

Ao longo deste documento foram descritas as principais decisões arquiteturais e de implementação de um serviço *peer-to-peer* de *timeline* descentralizado, onde as publicações de um utilizador são adicionadas à sua própria *timeline* e às *timelines* dos utilizadores que o seguem.

O serviço foi desenvolvido tomando em consideração os aspetos que julgávamos ser os mais importantes de explorar, como a garantia da ordenação causal das mensagens, a garantia de que um nodo ao voltar a *online* tenha acesso a todas as mensagens, e um cuidado extra para não sobrecarregar os nodos que geram mais ligações (utilizadores com mais seguidores), mediante a construção de hierarquias de nodos.

Julgamos ter explorado todos os aspetos que pretendíamos, tendo chegado a uma solução que resolve os problemas que nos propusemos a resolver, ainda que reconheçamos que num contexto real o serviço desenvolvido está longe de ser robusto o suficiente.

Como trabalho futuro, as hierarquias podiam ser melhoradas de forma a distribuir melhor o trabalho entre *peers* e também no sentido de diminuir as mensagens trocadas entre os mesmos, introduzidos pela redundância necessária para garantir que os nodos recebem as mensagens mesmo que os seus retransmissores estejam *offline*. Outro potencial problema são os problemas de segurança que não exploramos, por exemplo, o facto de as mensagens exporem o endereço IP de quem as enviou, mediante a análise do seu identificador. Isto seria facilmente resolvível usando um *hash* do IP, em vez do próprio. Para além disso, poderia-se adicionar criptografia de chave-simétrica para, mesmo que as mensagens possam ser intercetadas, dar-nos garantias que a mensagem foi, de facto, enviada por aquele *peer* e que só o *peer* destino consegue ler o seu conteúdo.

Referências

[1] Mayank Bawa, Brian F. Cooper, Arturo Crespo, Neil Daswani, Prasanna Ganesan, Hector Garcia-Molina, Sepandar Kamvar, Sergio Marti, Mario Schlosser, Qi Sun, Patrick Vinograd, and Beverly Yang. Peer-to-peer research at stanford. Technical Report 2003-38, Stanford InfoLab, September 2003.

[2] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design*, chapter 10. Addison-Wesley Publishing Company, USA, 5th edition, 2011.

[3] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 53–65, London, UK, UK, 2002. Springer-Verlag.

[4] Ryan King. Announcing snowflake. `https://blog.twitter.com/engineering/en_us/a/2010/announcing-snowflake.html`, June 2010.

[5] Boundary. Flake. `https://github.com/boundary/flake`. Repositório GitHub.