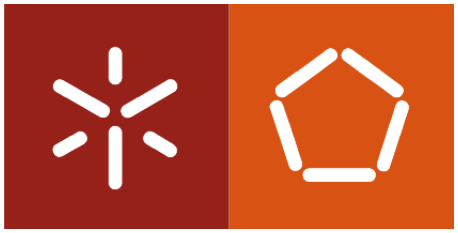

UNIVERSIDADE DO MINHO
Mestrado Integrado em Engenharia Informática

Sistemas Distribuídos em Larga Escala

Trabalho Prático - Relatório de Desenvolvimento

Jorge Oliveira (A78660)
José Ferreira (A78452)
Nuno Silva (PG38420)

19 de Maio de 2019

Conteúdo

Capítulo 1

Introdução e Exposição do Problema

As redes P2P estão em voga na atualidade. Têm a seu favor o facto de que cada elemento, em geral, pertencente a esta rede executa o papel de cliente e de servidor em simultâneo permitindo um maior escalabilidade, já que quantos mais *peers* mais força de trabalho a rede terá. São também ideais para difundir conteúdo para diversos utilizadores, fazendo com que o mesmo possa estar em diversos nodos não sobrecarregando um único servidor com todos os pedidos.

A nossa aplicação tem então por base uma rede P2P para partilha de estados de espírito de um utilizador, ou seja, é uma rede social onde uma pessoa pode partilhar com os outros curtos textos escritos pela própria. Cada utilizador pode ainda seguir outros para que daí em diante lhe seja apresentado conteúdo produzido pela pessoa que decidiu subscrever.

Uma estratégia para levar a que cada utilizador permaneça um maior espaço de tempo na rede (um dos problemas das redes P2P é a sua instabilidade) é o facto de quanto mais tempo o utilizador permanecer ligado, maior a quantidade de conteúdo e com mais rapidez o receberá. É ainda feito um *caching* de conteúdo de outros por parte de cada utilizador para facilitar a distribuição e divulgação dos textos para toda a rede de modo a que informação possa chegar de uma forma mais eficiente e rápida a todos os utilizadores com interesse na mesma.

Capítulo 2

Arquitetura do Sistema

Inicialmente o grupo definiu uma série de abordagens que lhes pareceram ser possíveis de implementar para o problema em questão. Como por exemplo a utilização da noção de *Super Peers* ou então de uma DHT. E por "cima"da camada de rede implementaríamos uma outra rede que iria tratar de todas as subscrições entre os diferentes *peers* podendo, por exemplo, optar por um sistema de hierarquias ou então através de um *flooding* controlado de mensagens.

De salientar que em qualquer uma das abordagens, previamente temos um nodo a correr que vai permitir que os nodos se liguem à rede P2P e que lhes passa a configuração correta para pertencerem à rede.

2.1 Possíveis Abordagens

Nesta secção serão apresentadas as abordagens sobre as quais o grupo discutiu e que, após analisadas, decidiu não implementar porque não as mais vantajosas.

2.1.1 *Super Peers*

Nesta abordagem existe a noção de que um nodo poderá desempenhar um papel diferente dos outros tendo uma maior carga de trabalho. Tínhamos duas opções : um dispositivo "normal", após analisadas as suas caraterísticas poderia ser promovido a *Super Peer* ou então tínhamos entidades já a correr que pertenceriam sempre à rede mas que nunca seriam dispositivos móveis, ou seja, eram nodos totalmente controlados e geridos por nós. Caso optássemos pela primeira opção teríamos a dificuldade em conseguir eleger um nodo "normal"para *Super Peer* pois teria de ser um nodo que nos desse garantia de que se iria manter por muito tempo na rede. Na segunda opção temos a certeza que o *Super Peer* tem definitivamente uma capacidade acima dos outros e que permanecerá para sempre na rede mas depois podemos ficar com vários problemas ao nível de escalabilidade.

No entanto a ideia fundamental é que os nodos *Super Peer* iriam formar uma rede entre si (sem que seja obrigatório que todos tenham uma ligação para todos) e que posteriormente cada nodo se deveria de ligar a um e apenas um nodo *Super Peer* sendo que a carga deveria estar distribuída para evitar que um deles ficasse com demasiados utilizadores. Então cada nodo *Super Peer* era responsável por gerir um conjunto de nodos "normais"bem como guardar a informação sobre que utilizadores cada um deles seguia para redirecionar as mensagens que lhes poderiam interessar. Desta forma sempre que um utilizador subscrevia outro essa informação era passada para o *Super Peer* e informava os restantes para que sempre que existisse uma mensagem do dito utilizador ele teria de recebê-la para depois enviar para os utilizadores interessados.

Verificámos que potencialmente sempre que era enviada uma mensagem ela teria de circular por todos os nodos *Super Peer* que por sua vez teriam de manter uma lista de que para cada utilizador quais os utilizadores que os mesmos seguiam. O que poderia diminuir bastante a eficiência na distribuição de mensagens e ainda aumentaria a complexidade de gestão de subscrições que um nodo *Super Peer* teria. O lado bom desta abordagem é que diminuiria a complexidade para um nodo normal já que apenas teria de comunicar com o *Super Peer*. Poderíamos ainda optar por estabelecer comunicações entre os diferentes *peers* no entanto verificámos que tal ideia não faria muito sentido.

2.1.2 Hierarquia de Subscrições

Esta estratégia pressupõe que já possuímos a rede P2P toda formada e que apenas iria gerir o sistema de subscrições e também de propagação de mensagens. Um grande problema é se existe um utilizador bastante popular, isto é, com muitos seguidores. Neste caso, e se optássemos por uma estratégia de que quando um utilizador segue outro cria uma ligação para o mesmo este *outlier* iria ficar com demasiadas conexões não conseguindo dessa forma executar a aplicação. Desta forma pensámos num sistema de hierarquia semelhante a uma árvore (só não seria uma árvore porque iríamos implementar redundância de caminhos, pelo que seria um grafo).

Quando um utilizador segue outro, através dos seus ids, era calculada uma distância (semelhante ao caso do Kademlia, ou seja, uma distância xor). Caso ficasse num determinado intervalo então iria conectar-se diretamente a ele, senão teria de se conectar a um dos outros utilizadores que o seguiam. No entanto, como é uma rede P2P, há uma grande instabilidade na mesma já que um nodo pode estar sempre a entrar e a sair da rede, pelo que esta hierarquia iria estar sempre a ser recriada para garantir que todos os nodos têm acesso às mensagens do utilizador. Devido a esta complexidade que traria uma enorme ineficiência na aplicação bem como uma maior dificuldade para garantir que todos os nodos recebiam a mensagem decidimos não optar por esta estratégia. Teoricamente pareceu-nos uma ideia exequível, no entanto na prática pareceu-nos algo impossível e bastante ineficiente pela gestão de toda a arquitetura e as constantes mudanças.

2.1.3 Ordem das mensagens

Relativamente à ordem das mensagens pensamos em ter uma versão de causalidade. Esta não poderia, no entanto, ser com total conhecimento do sistema visto que o objetivo era exatemente não conhecer todo o sistema. Pensamos então numa ideia de ter causalidade apenas entre os seguidores, fazendo por exemplo interseções entre os utilizadores que conhecíamos, contudo reparamos que esta ideia não funcionaria corretamente.

Decidimos então focarmo-nos noutros aspectos, como por exemplo a arquitetura da solução, e apenas garantimos a ordem de mensagens da seguinte forma:

- se forem do mesmo utilizador são ordenadas por id de mensagem
- senão são ordenadas por data (temporalmente)

2.2 Abordagem Implementada

A abordagem escolhida para o sistema a implementar foi o uso de uma DHT (Kademlia) como base do sistema. Quando um nó se pretende ligar ao sistema, é ligado à rede gerida pelo Kademlia. Usamos a DHT para guardar a associação entre username e a sua localização na rede. Isto permite que qualquer utilizador consiga descobrir outro na rede e comunicar com este. Por cima desta rede gerida pelo Kademlia é usado um protocolo de *gossip* que é responsável por permitir a propagação das publicações pela rede.

2.2.1 Propagação de Mensagens

Informação de uma publicação

Uma publicação contém a seguinte informação:

- *username* - o identificador do utilizador que a realizou
- id - o identificador da mensagem
- conteúdo - o conteúdo da publicação
- data - a data da publicação
- *timestamp* - um identificador temporal que indica a validade da publicação

Como guardamos um identificador temporal para indicar a validade da publicação podemos ter um problema quando um utilizador sai por exemplo num fuso horário de Portugal e entra com o relógio no fuso horário da China, que já se encontra adiantado, descartando a publicação que tinha guardado no seu ficheiro local. Possivelmente podia ser resolvido se tivéssemos alguma forma de perceber quanto tempo o utilizador estaria *offline* ou então se tivéssemos uma conversão entre fusos horários.

Propagação

Para a propagação de mensagens utilizamos uma estratégia desestruturada de *gossip* que é bastante simples. Sempre que um utilizador publica uma mensagem seleciona um número de seguidores aleatórios para enviar a mesma e indica quais os que faltam receber. Sempre que um utilizador recebe uma mensagem de publicação pela primeira vez, seleciona também um conjunto de utilizadores aleatórios dos que faltam e envia para esses, acabando esta propagação quando os utilizadores tenham todos recebido a mensagem pela primeira vez.

Ao realizar alguns testes para verificar se este protocolo era fiável verificamos que o mesmo podia falhar, pelo que realizamos uma pesquisa e chegamos à conclusão que ao selecionar os seguidores aleatórios era necessário selecionar um número que nos garanta alguma segurança. Segundo o artigo *"Cuckoo: Towards Decentralized, Socio-Aware Online Microblogging Services and Data Measurements"*[1], verificamos que o melhor era enviar para: ln(N)+K, sendo que N era o número de utilizadores a que a mensagem devia chegar e K um fator de replicação que garantia que a mensagem era entregue a todos os utilizadores com uma probabilidade de $e^{-e^{-k}}$. Definimos o K como 10 garantindo assim que chega a todos em 99.995% das vezes, o que nos pareceu ser suficiente.

De seguida vamos apresentar alguns resultados relativos aos testes realizados. De referir que realizamos apenas 50 tentativas para cada um do número de utilizadores a que a mensagem teria de chegar pelo facto de não termos grande capacidade de processamento.

(a) *Relative message redundacy*

(b) Mensagens por número de seguidores

Figura 2.1: Testes realizados

Podemos verificar na figura 2.1b que o número de mensagens cresce linearmente com o número de seguidores. Verificamos ainda na figura2.1a que o *relative message redundacy* se encontra entre valores de 14 e 16 e é crescente o que indica que o protocolo impõe algum *overhead* na rede. No entanto o número de rondas necessárias para chegar a todos os seguidores andou sempre por volta de 4/5 rondas.

Através dos testes realizados observa-se que o protocolo impõe alguma sobrecarga em termos de mensagens na rede, no entanto necessita de poucas rondas e como foi referido anteriormente é bastante fiável. Algo que podia ser feito é uma análise mais extensa entre qual o melhor *trade-off* entre número de mensagens (RMM), fiabilidade e número de rondas para chegar a um valor ótimo.

2.2.2 Pedidos de Timeline

Sempre que um utilizador volta a estar ativo tem de voltar a pedir a *timeline* de todos os utilizadores que segue, para que a possa ter atualizada e sirva de *caching* para pedidos de outros participantes. A estratégia que definimos para obter a *timeline* de um usuário foi pedir a *timeline* a um utilizador de cada vez. Neste pedido é ainda facultado a *timeline* dos utilizadores que os dois utilizadores seguem em comum. Desta forma podemos reduzir o número de

pedidos efetuados bem como o número de conexões criadas. Existe uma grande probabilidade de ao seguirmos um determinado utilizador A e outro B, que o utilizador A também siga o B.

O pedido de *timeline* é realizado a um utilizador de cada vez seguindo a seguinte ordem: primeiro pedimos a todos os utilizadores que estamos a seguir, caso continuem a existir utilizadores que ainda não estejam atualizados, por exemplo porque se encontravam inativos, pedimos a *timeline* aos seguidores desses utilizadores até que tenhamos a *timeline* totalmente atualizada.

Como estamos a trabalhar com *timestamps* para verificar se uma dada publicação já se encontra desatualizada e como não existem garantias que todos os utilizadores tenham o relógio sincronizado, sempre que enviamos uma resposta a um pedido de *timeline* colocamos também o nosso *timestamp* atual para que o utilizador saiba quanto tempo de vida é que as publicações ainda têm. Continuamos com um problema já que não contamos com o tempo de envio da mensagem, no entanto este não se compara ao de existirem relógios totalmente dessincronizados, visto que este tempo de envio é normalmente reduzido e pode ser desprezado.

2.2.3 Informações Guardadas

DHT

Como já foi descrito anteriormente a informação que guardamos na *DHT* é apenas meta-informação relativa a cada um dos utilizadores e serve para garantir a conectividade entre os mesmos. Guardamos o nome do utilizador e o endereço onde aceita conexões, para além disso guardamos também informações relativas aos seus seguidores, importante para poder ser seguido por outros utilizadores quando se encontra *offline*.

- username
- (ip,porta)
- seguidores - informações relativas aos seguidores do utilizador em questão, nomeadamente:
  - username
  - (ip,porta)

Localmente

Para além disto cada utilizador mantém localmente alguma informação relativa a si próprio, o seu id, o id da última mensagem que enviou, a sua *timeline*, *following_timeline* - a timeline dos utilizadores que está a seguir (apenas as publicações que ainda não expiraram) de modo a poder servir como *cache* dos mesmos. Guarda também algumas informações sobre os utilizadores que está a seguir, nomeadamente:

- username
- (ip,porta) - importantes para quando tem de pedir a timeline
- id da última mensagem recebida - importante para garantir a ordem na receção de mensagens

2.2.4 Descoberta de novos utilizadores

Para seguir um utilizador, é necessário conhecer o seu *username*. Para permitir que os utilizadores do sistema possam descobrir utilizadores sem precisarem de adivinhar o seu *username*, o sistema disponibiliza uma opção de descoberta de novos participantes no mesmo. Quando um utilizador quer descobrir novos utilizadores para seguir, são escolhidos alguns seguidores de utilizadores que este já segue e são apresentados os seus *usernames*. Um exemplo: O João segue o Manuel e a Maria segue o Manuel. Se o João quer descobrir novos utilizadores, o sistema indicará o *username* da Maria para o João poder seguir. Caso não seja encontrado um número mínimo de utilizadores nesta procura, são visitados os seus próprios seguidores e aplicado o mesmo princípio descrito acima. Se mesmo assim não conseguir atingir o número mínimo, por fim percorre os vizinhos do utilizador na rede gerida pelo Kademlia.

Capítulo 3

Problemas e Trabalho Futuro

O grupo está ciente de que a solução por nós implementada não é a ideal, aliás é muito complicado arranjar uma solução ideal para um problema como o que enfrentamos. As redes P2P são bastante apetecíveis por permitirem que o conteúdo fique distribuído por todos os nós da rede o que aumenta consideravelmente a escalabilidade do sistema, no entanto apresenta muitos problemas e é complicado conseguir corrigi-los todos. Temos, portanto, noção dos problemas que estão por detrás da nossa implementação e vamos passar a expô-los.

Utilizamos uma DHT para guardar meta informação sobre um utilizador e para nos garantir a ligação entre os diferentes *peers* da rede. Na implementação que utilizamos, se quiséssemos inserir informação então tínhamos a operação de *set* e caso quiséssemos retirar informação da DHT utilizávamos a operação de *get*. O problema por detrás destas duas operações reside no facto de que dois utilizadores podem fazer um *get* simultâneo e em seguida cada um efetua um *set*, neste caso uma das informações irá ser perdida. Um exemplo deste caso é tanto o utilizador B e C querem seguir o A, mas para o fazerem têm de fazer *get* da informação do A e depois *set*. Como realizaram o *get* concorrentemente (sem nenhum fazer *set*) então apenas um ficou como seguidor do A. É uma perda fantasma de informação. Para resolver este problema podíamos alterar a implementação do *Kademlia* para que pudéssemos aplicar uma função a um determinado conteúdo, em vez de ter de fazer *get* e *set*. Por exemplo se aplicássemos a função de acrescentar um elemento a uma lista, mesmo que fosse efetuado por 2 utilizadores concorrentes, esta não se perdia pelo facto de apenas ser um *append* para cada um.

Cada nodo guarda alguns dados no Kademlia e estes não se encontram noutros nodos. Caso um nodo saia da rede e perca completamente a conectividade com todos os outros então poderá ser possível que aquela informação se perca porque não se encontra em mais nenhum nodo e também não foi possível efetuar a transferência da mesma. Uma forma de ultrapassar esta contrariedade era alterar a implementação do Kademlia de modo a introduzir redundância, para que a informação pudesse ficar guardada por mais do que um nodo de forma a termos um *backup* da mesma por toda a rede.

Um utilizador, quando fica *online* realiza pedidos para atualizar a *timeline* aos outros utilizadores. Porém pode não ser possível obter as mensagens de um determinado utilizador já que o mesmo se pode encontrar *offline* e mesmo utilizando a estratégia de pedir aos outros *users* a *timeline* do que se encontra inativo podemos não a conseguir obter se todos os outros também estão *offline*. Este é um caso raro, mas que merece atenção e podia ser resolvido se tivéssemos uma noção de um *super-peer* que não é um utilizador real do sistema mas que pode seguir alguns utilizadores aleatórios (baseado em algum critério pré-definido) para servir de *cache*. Esta solução permitiria uma maior fiabilidade pelo facto de este *super-peer* ser controlado por nós e ser um "utilizador"com maior poder de processamento e estar constantemente ativo.

Outro problema deve-se ao facto de utilizarmos relógios físicos para marcar as mensagens o que faz com que cada *peer* possa ter relógios diferentes. O ideal seria que todos os utilizadores tivessem os relógios sincronizados para que a nossa implementação neste aspeto funcionasse perfeitamente. Outra forma seria implementar causalidade através de vetores de relógios, mas como mencionado anteriormente era difícil conseguir executar esta ideia dado que é necessário que cada *peer* conheça todos os outros elementos da rede, algo que não é viável.

Para garantir ordem nas mensagens de um mesmo utilizador utilizámos identificadores únicos que são sequenciais e quando recebemos uma mensagem só a aceitamos caso a mesma possua o número de sequência correto. Isto poderá fazer com que não se aceitem novas mensagens. Imaginemos o seguinte cenário, o utilizador A recebeu uma nova

mensagem do utilizador B com o *id* com o valor de 5, no entanto a última mensagem que possuía do mesmo tinha o *id* 3, ou seja, estava à espera da mensagem com o *id* 4. Caso a mensagem tenha sido perdida durante a circulação da mesma pela rede fará com que o utilizador A não aceite novas mensagens do utilizador B. Isto poderia ser resolvido com um pedido de forma ativa por parte do A para lhe enviarem de novo a mensagem com o *id* 4. Este pedido podia ser efetuado a algum nodo que siga o utilizador B ou mesmo o próprio utilizador B. Outra forma poderia ser definir um *timeout* onde caso não recebêssemos a mensagem então passávamos a sequência à frente e aceitávamos as mais recentes, desta forma impedíamos que não existisse uma evolução no utilizador A sobre o utilizador B, mas estando cientes de que existiu uma inconsistência na informação que o utilizador A teve direito.

Este problema de receber uma mensagem e aceitar sem antes receber outra pode ser crítico ainda para os pedidos de *timeline* visto que um utilizador pode achar que recebeu uma *timeline* de outro que está atualizada e pode não estar se não for realizado qualquer tratamento deste caso.

Uma outra forma possível de abordar estes casos tanto de causalidade, como de ordem de mensagens seria considerar que existiam publicações e podiam existir respostas a essas publicações. Nesses casos a causalidade estava implícita no caso de uma resposta ter uma causa na publicação e a mesma não poderia aparecer antes da publicação. De resto a ordem de elementos do mesmo nível poderia ser como foi referido no início.

Para o sistema de subscrições utilizámos o protocolo de transporte TCP. Sabemos que o mesmo nos dá muitas garantias no entanto sempre que dois utilizadores se conectam criam realmente uma conexão entre os mesmos. Apesar de que sempre que efetuámos uma conexão para enviar uma mensagem em seguida a fechamos temos a noção que num espaço curto de tempo um utilizador possa ter demasiadas ligações e poderá não aceitar mais nenhuma não conseguindo desta forma conseguir enviar ou receber toda a informação de que necessita. Mas isto poderá acontecer esporadicamente pelo que não impede o funcionamento correto da nossa aplicação. Outro problema a considerar é o facto de poderem existir *firewalls* que impeçam ligações diretas entre dois utilizadores, nestes casos o ideal seria arranjar nodos intermédios e propagar a informação para que então indiretamente os dois utilizadores possam estabelecer uma ligação, no entanto a complexidade aumentaria.

Bibliografia

[1] Cuckoo: Towards Decentralized, Socio-Aware Online Microblogging Services and Data Measurements Tianyin Xu, Yang Chen, Jin Zhao, Xiaoming Fu Institute of Computer Science, University of Goettingen, Goettingen, Germany State Key Lab. for Novel Software and Technology, Nanjing University, Nanjing, China School of Computer Science, Fudan University, Shanghai, China