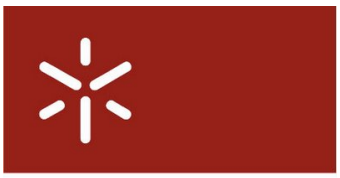
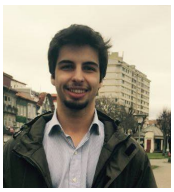


Universidade do Minho

Sistemas Distribuídos em Larga Escala

MIEI - 4º ano - 2º semestre
Universidade do Minho

Relatório
Decentralized Timeline


Dinis Peixoto
A75353


Ricardo Pereira
A74185


Marcelo Lima
A75210

16 de Junho de 2018

1 Introdução

O presente relatório abordará toda a conceção e implementação de um serviço de *timeline* descentralizado numa rede de dispositivos *peer-to-peer*. Cada utilizador representará um nodo nesta rede e deverá ser possível que estes publiquem mensagens, subscrevam outros utilizadores, passando a receber as mensagens que estes publicam e ainda auxiliando a propagação destas para outros utilizadores. O conteúdo remoto deve estar disponível quando uma fonte, ou um subscritor desta esteja *online* e, por isso, consiga propagar a informação. A informação recebida pode ser guardada apenas durante um determinado período de tempo.

2 Descrição do problema

A conceção de um serviço descentralizado trás inúmeras vantagens e com isso inúmeros problemas que devem ser abordados e resolvidos para uma correta implementação do mesmo. Numa primeira abordagem com o trabalho prático o grupo, durante uma sessão de *brainstorming*, levantou uma série de problemas que deveriam ser discutidos e eventualmente resolvidos.

O primeiro problema surgiu logo na conexão de um novo nodo à rede P2P, este tem, de algum modo, saber a quais dos restantes se pode conectar. Lembrando que, considerando a inexistência de um servidor centralizado deixa também de haver um ponto único de acesso à rede.

Partindo agora do princípio que temos uma rede formada, uniformemente distribuída, isto é sem aglomerados, passaríamos para situações mais complexas, como a difusão de uma mensagem. Assumindo um elevado número de utilizadores, a publicação de uma mensagem por parte de um utilizador não poderia ficar completamente a cargo do mesmo, caso contrário não estaríamos a tirar proveito da rede na qual nos inserimos. Consideremos o exemplo da *Katy Perry* com cerca de 100 milhões de seguidores no Twitter, a publicação de um *post* demoraria horas de processamento até que este fosse entregue a todos os seus seguidores. Neste ponto, o grupo ponderou diversas estratégias, sempre com o foco de repartir o trabalho de difundir uma mensagem por vários nodos e não só pelo nodo fonte da mesma.

Uma situação idêntica seria quando um utilizador inicia sessão (passa a *online*) e deve pedir às pessoas que segue as suas timelines para acertar a sua conforme as mesmas. O caso da *Katy Perry* uma vez mais traria uma situação complicada visto que estaria regularmente a receber pedidos dos seus *followers* para enviar a sua timeline, o que por vezes pode nem ser necessário se as timelines dos utilizadores já estiverem devidamente atualizadas, se, por exemplo, a *Katy Perry* não fez nenhuma publicação enquanto o utilizador estava *offline*.

Por fim, as timelines devem ser apresentadas por ordem cronológica, não devem ser permitidas mensagens vindas do futuro, ou que não encaixem no espaço temporal considerado na timeline de um utilizador.

3 Primeira abordagem: Super-peers

Inicialmente, quando nos debruçamos pela primeira vez na tentativa de encontrar uma solução para o problema apresentado, chegamos a uma resposta que vale a pena aqui mencionar. Baseando-nos bastante na arquitetura do Kazaa, chegamos à conclusão que iríamos optar pelo uso de supernodes.

Assim, inicialmente, na rede existiria um servidor de bootstrap no qual os novos nodos se iriam ligar. Quando um nodo atingisse um x número de ligações iria-se transformar em supernode. Quando um supernode ficasse *offline*, este iria escolher aleatoriamente um dos nodos-filho, o qual passaria a assumir o papel de supernode, herdando a sua hash table. Cada supernode iria armazenar uma hash table com informação dos followers de cada nodo conectado a ele (nodos-filho).

Ora, quando um nodo publicasse uma mensagem, esta iria ser redirecionada para o seu supernode que reencaminharia a mesma para os restantes supernodes. Nos supernodes, estes verificariam se algum dos seus filhos necessitava de receber essa mensagem, consultando a hash table. Se a resposta fosse positiva, a mensagem era reencaminhada até chegar ao nodo correspondente. Sempre que um node ficasse novamente *online* na rede, iria pedir ao seu supernode para localizar os nodos que ele se encontrava a seguir ou os seguidores do mesmo, de modo a estabelecer uma ligação direta com estes para receber as respetivas timelines.

4 Segunda abordagem: Distributed Hash Table

O grupo não optou pela abordagem anterior por uma série de razões, sendo a principal a inexistência de uniformidade entre a distribuição dos nodos, uma vez que inevitavelmente se criariam aglomerados em torno dos super-nodos. A estratégia inicial pensada exigiria que os super-peers comunicassem entre si para saber, por exemplo, onde podemos encontrar um determinado nodo para lhe pedir a timeline, este pedido para além de ser difundido por todos os super-nodes geraria o envio de muitos pedidos inúteis. Além disto, todas as mensagens postadas na rede passariam pelos super-peers, o que implicaria sobrecarga nos mesmos, independentemente do rácio de $\frac{supernode}{node}$ da rede.

Posto isto, seguimos para a abordagem seguinte: a utilização de uma *Distributed Hash Table* - garantindo com esta que cada nodo da rede contribuí equitativamente e que, jogando com as ligações entre estes, conseguimos repartir o uniformemente o esforço realizado por cada um destes para as diferentes funcionalidades permitidas.

Com esta decisão viria uma outra, que seria qual das Distributed Hash Tables lecionadas utilizar: *Chord* [1] ou *Kademlia* [2]. Não foi, de todo, uma decisão complicada, uma vez que a implementação do *Chord* não se aplicava para o serviço que pretendíamos implementar.

Relativamente à utilidade que a distributed hash table nos irá fornecer, esta passa pela mesma que foi decidida na primeira implementação relativa à hash table de cada supernode. Assim, as entradas da DHT serão preenchidas com a informação dos seguidores de cada utilizador, assim como a sua *Port* e *IP* e, ainda um *vetor clock* para controlar a causalidade. Para cada entrada da hash, a Key

corresponde ao id do utilizador e o Value à informação anteriormente descrita em formato `JSON`, seguindo a seguinte estrutura:

```
{
    "ip": "192.168.1.88",
    "port": "5050",
    "followers": {
        "user2": "192.168.1.87 7071"
    },
    "vector_clock": {
        "user1": 4,
        "user2": 3
    }
}
```

Figura 1: Exemplo de uma entrada na DHT

5 Implementação

Na presente secção abordaremos as estratégias utilizadas na implementação do serviço final que, embora não totalmente completo, consegue representar grande parte das metodologias, algoritmos e soluções refletidas pelo grupo de modo a resolver os problemas inicialmente apresentados.

5.1 Arquitetura da aplicação

Depois de várias trocas de ideias e, tendo em conta o que foi anteriormente dito, finalmente chegamos a uma arquitetura final da nossa rede descentralizada.

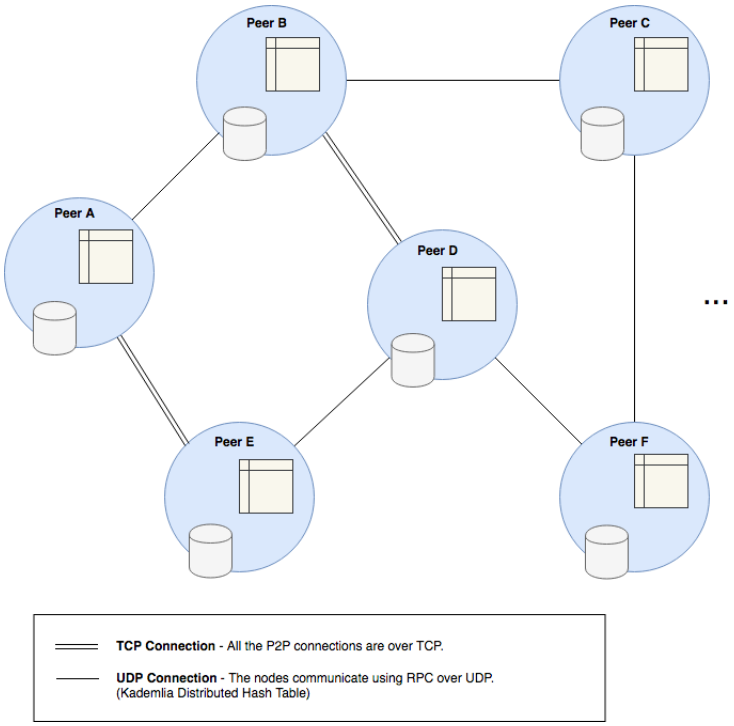


Figura 2: Arquitetura da rede

Como podemos observar pela figura, os nodos estabelecem entre si dois tipos de conexões: TCP e UDP. As conexões TCP correspondem às ligações P2P usadas para trocar mensagens síncronas específicas entre os nodos. Já as ligações UDP são necessárias para garantir a distribuição correta da hash table por todos os nodos do sistema, através de pedidos/mensagens assíncronas.

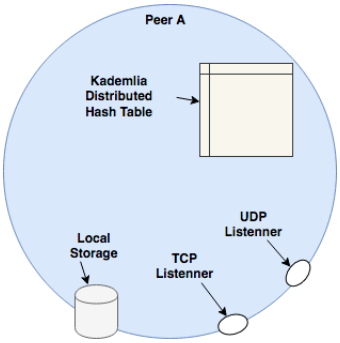


Figura 3: Exemplo de um nodo

Cada nodo por si só tem um conjunto de componentes que é importante notar. A presença de uma hash table distribuída permite que seja acedida por cada um dos nodos presentes na rede e, em simultâneo, que cada um destes seja responsável por armazenas uma porção da mesma. Num nodo existem duas portas em escuta, uma para receber ligações TCP e outra para receber ligações UDP. Por último, existe um conjunto de dados persistidos em memória (*local storage*): uma lista com informação dos utilizadores que um utilizador está a seguir e ainda uma lista com todas as mensagens do próprio - esta é persistida localmente.

5.2 Funcionalidades da aplicação

De modo a ir de encontro ao que nos foi pedido no enunciado do trabalho, tivemos de implementar três funcionalidades essenciais na nossa rede descentralizada.

5.2.1 Difusão de mensagens

Todos os utilizadores terão ao seu dispor a capacidade de inserir uma mensagem na sua timeline. Quando este evento acontece, depois da aplicação receber a mensagem inserida pelo utilizador, decorre um conjunto de processos essenciais para o correto funcionamento de todo o nosso sistema.

O primeiro passo passa por guardar a mensagem localmente para que esta nunca seja perdida e possa ser transmitida para outros utilizadores quando necessário. De seguida é preciso atualizar o vector clock do utilizador, ou seja, incrementar o seu valor correspondente.

Esta mensagem é enviada a apenas uma dada percentagem dos seus seguidores. Estes utilizadores, ao receber a mensagem vão reencaminhar a mesma para uma percentagem dos seguidores do utilizador fonte, mas apenas para aqueles que ainda não a receberam, para isso verifica primeiro o estado do vector clock do utilizador e só se este não estiver atualizado, ou seja, se ainda não recebeu a mensagem, é que processe para o envio da mesma. Isto é repetido sucessivamente até todos os seguidores receberem a mensagem.

5.2.2 Seguir um utilizador

Esta é uma funcionalidade bastante importante, pois sem ela as timelines dos utilizadores seriam unicamente locais e construída independentemente das dos restantes utilizadores.

Quando um utilizador demonstra intenção de seguir um outro utilizador, tem de fornecer o id correspondente. A aplicação, após receber o id consulta a hash table de forma a verificar se esse utilizador existe. Caso este exista, vai atualizar a sua entrada na tabela, acrescentando aos seus seguidores a informação do seu novo seguidor e ainda colocar o valor do vector clock correspondente a 0.

5.2.3 Mostrar *timeline*

A timeline de um utilizador é a junção das timelines de todos os utilizadores que ele segue juntamente com a sua. Assim, quando um utilizador inicia sessão na aplicação, esta vai percorrer todos os utilizadores que este segue e verificar se os seus valores do vector clock estão em conformidade com os que ele tem no momento. Caso não estejam, dada a lista dos seguidores desse utilizador, é selecionado aleatoriamente um deles que apresente um vector clock mais atualizado que o seu e é-lhe enviada uma mensagem de pedido de timeline. Quando este recebe o pedido, envia as mensagens correspondentes ao intervalo calculado pela diferença entre o seu vector clock e o do emissor. De seguida, reencaminha esse mesmo pedido para outro seguidor, seguindo a mesma regra descrita anteriormente. Isto é repetido até que o vector clock do emissor fonte fique totalmente atualizado.

5.3 Ferramentas

A implementação do serviço final exigiu a utilização de algumas ferramentas indispensáveis para o seu desenvolvimento. A linguagem optada foi o `Python` uma vez que é uma linguagem com a qual o grupo se sente à vontade e que permite a integração de módulos auxiliares à implementação do mesmo. Como módulos auxiliares principais utilizamos o `Asyncio` [3] e o `Kademlia` [4], para realizar pedidos assíncronos e abstrair a implementação da distributed hash table (*Kademlia*), respectivamente. É de notar que a utilização do Asyncio foi necessária uma vez que todos os pedidos realizados à DHT são assíncronos.

6 Problemas por resolver

Apesar de termos uma arquitetura pensada ao pormenor somos capazes de reconhecer que esta não é, de todo, perfeita e ainda apresenta algumas falhas que teriam eventualmente de ser corrigidas.

- Tempos físicos - durante toda a idealização da arquitetura optou por se abstrair dos tempos físicos, uma vez que máquinas/utilizadores distintos podem ter fusos horários distintos. O grupo focou-se apenas em garantir ordenação causal entre as mensagens enviadas por cada utilizador através do armazenamento de um *vector clock*.

- Sobrecarga na DHT - tanto a difusão de mensagens como a atualização do *vector clock* quando um utilizador volta a estar *online* são demasiado dependentes da *distributed hash table*, gerando muitos pedidos à mesma. Como esta não foi implementada pelo grupo e ainda considerando que a sua implementação é feita utilizando pedidos assíncronos, assumiu-se que apresentava uma excelente *performance* tanto ao fazer `get` (consulta) como ao fazer `set` (atualização) da informação. Num cenário real isto pode não acontecer e os pedidos excessivos à mesma podem levar a uma considerável redução da performance de toda a rede.

- Difusão de mensagens - o método utilizado para a difusão de mensagens foi o que gerou mais controvérsia entre o grupo. Uma solução inicial foi que cada utilizador criava uma conexão P2P com cada um dos seus *followers* para publicar uma mensagem, isto é, para garantir que esta iria chegar a todos os seus seguidores, no entanto, esta solução não é nada escalável. A solução final passou pela difusão da mensagem pela rede, dividindo este trabalho por vários nodos. Cada nodo fica responsável por reenviar a mensagem por outros x nodos, mas antes de enviar verifica se este já recebeu ou não a mensagem, deste modo, ao fim de algumas iterações teríamos a certeza que a mensagem chegava a todos os seguidores do nodo inicial. Esta metodologia tem, no entanto, um problema: os utilizadores *offline* seriam deixados para o fim e todos os restantes utilizadores acabariam a tentar enviar-lhes a mensagem, uma vez que o *vector clock* destes não estaria atualizado conforme o devido. Para solucionar este problema seria necessário pensar numa estratégia, por exemplo, um *timeout*, para que os restantes nodos desistissem de tentar enviar a mensagem.

Referências

[1] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications
`pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf`

[2] Petar Maymounkov and David Mazières. Kademlia: A Peer-to-peer Information System Based on the XOR Metric
`pdos.csail.mit.edu/p̃etar/papers/maymounkov-kademlia-lncs.pdf`

[3] Asyncio: Asynchronous I/O, event loop, coroutines and tasks,
`docs.python.org/3/library/asyncio.html`

[4] Kademlia: Distributed Hash Table in Python using asyncio,
`kademlia.readthedocs.io/en/latest/`

[5] TCP/IP Client and Server,
`pymotw.com/3/socket/tcp.html`

[6] Socket Programming,
`docs.python.org/3/howto/sockets.html`