Pru - decentralized timeline

Carlos Silva A75107
Catarina Cardoso A75037
José Silva A74601

Universidade do Minho, Braga, Portugal

Resumo Define-se por *microblogging* o serviço *online* que permite enviar atualizações rápidas usando pequenas frases, imagens ou endereços de videos, relacionando os conceitos de blogue e de rede social. *Pru* é um *sofware* de *microblogging* que se difere das restantes redes sociais por ser descentralizado. A utilização de uma rede não centralizada não obriga à aquisição de servidores centrais que funcionem em larga escala e não expõe os dados de todos os utilizadores a qualquer empresa, trazendo mais escabilidade (a baixo custo) e privacidade.

1 Introdução

O projeto foca-se no objetivo da criação de uma *timeline* descentralizada explorando *peer-to-peer* e *edge devices*. Este serviço permite que utilizadores possam subscrever outros e desta forma, ao verem o que estes publicam, armazenam também as suas publicações, além das próprias, nas máquinas pessoais. Haverá, no entanto, um servidor para que se armazene um catálogo associando um nome de utilizador a um endereço IP, lidando com as mudanças de localização das máquinas dos utilizadores. Além de catalogar utilizadores armazena quais as subscrições existentes no serviço.

2 Trabalhos relacionados

O ***Gnu Social*** [1] é um *software* de *microblogging* descentralizado que permite aos seus utilizadores realizarem comunicações públicas e privadas. Está implementado para ajudar comunidades ou empresas a trocarem pequenas atualizações de estado, fazerem votações e anunciarem eventos. Nesta rede social existe o conceito de *instance/service*, ao qual os utilizadores se devem associar, podendo comunicarem na mesma com outros utilizadores que não se encontrem nessa *instance*. As conexões entre *instances* são estabelicidades quando utilizadores de diferentes *instances* se conectam.

Embora o gnu social não seja a mesma coisa que se pretende projetar, envolve o conceito de uma rede social descentralizada. Isto faz com que se relacione com o trabalho, pois o objetivo é similar.

O ***Diaspora**** [2] designa-se por ser uma rede social baseada em descentralização, liberdade e privacidade. Os novos utilizadores devem seleccionar a que *pod* se pertendem ligar ou então podem-se inscreverem para hospedar um *pod* na sua máquina. Um *pod* trata-se de um servidor independente que se conecta a outros *pods* existentes, realizando assim as comunicações entre todos os utilizadores.

Com esta arquitetura descentralizada, o *Diaspora** afirma não existerem os problemas presentes numa rede centralizada gerida por uma empresa: em que os dados dos utilizadores podem ser pirateados ou perdidos e que qualquer problema do servidor central pode provocar lentidão ao serviço.

Newton [3] trata-se de um *Twitter* descentralizado que funciona como um gerador local de ficheiros *JSON* que são endereçados publicamente. Existe um módulo de armazenamento para publicações locais que requer um *web server* e outro que utiliza o serviço *Google Drive*. Como cada utilizador tem um ficheiro com uma lista dos utilizadores que segue, quando pretende atualizar a sua *timeline*, são resgatadas as publicações mais recentes dos utilizadores presentes nessa lista.

3 Decisões de design e desafios

3.1 Publicações

As publicações dos utilizadores serão mensagens de 280bytes, no máximo. Esta limitação do tamanho das mensagens tem como objetivo evitar que as mensagens comprometam a memória do utilizador demasiado rápido. Cada mensagem terá um id, mas este é único apenas para cada utilizador, ou seja, não é possível identificar uma mensagem apenas pelo seu id. Esta forma de identificação das mensagens será útil nos pedidos de novas mensagens.

3.2 Memória

A memória limite é um fator a considerar neste sistema, visto que se trata de memória dos utilizadores. É por isso importante limitar a memória utilizada por cada peer. Esta limitação só se aplica a dados provenientes de utilizadores que este subscreveu. O objetivo é não gastar demasiada memória sobre a qual o utilizador não tem controlo (não consegue apagar este tipo de informação). As publicações do utilizador ficam na sua máquina para sempre, a não ser que as apague manualmente. Para as publicações dos subscritores haverá um *timeout* em cada mensagem guardada. Este *timeout* será de 1 semana, para que os seus seguidores possam aceder apenas aos *posts* mais recentes. No caso em que não há memória para guardar mais mensagens (de outros), serão apagadas as mais antigas e substituídas pelas novas, mesmo que ainda não tenha passado o tempo de *timeout*.

3.3 Servidor de catálogo e criação de uma rede conectada

Haverá um servidor central que servirá apenas de catálogo para obter os IPs das máquinas ligadas. Quando uma máquina avisa este servidor central que se ligou, recebe uma confirmação juntamente com os IPs de alguns utilizadores escolhidos aleatoriamente para se ligar, ficando conectado à rede. Por uma questão de eficiência existirá um limite de 10 conexões por *peer*, isto pode fazer com que um (ou mais) dos peers fornecidos pelo servidor para conexão não consiga aceitar a mesma. Para resolver esta questão, sempre que isto acontece, o *peer* sobrecarregado sugere um dos seus vizinhos à nova conexão. Mais importante que o número máximo de conexões, é o número mínimo de conexões que cada *peer* tem que manter, de forma a existir uma grande probabilidade da rede ser uma *connected component*. Um grafo d-regular com $d \geq 3$ é conectado assintoticamente, quase de certeza [4]. Sabendo isto, é intuitivo que a probabilidade de ter um grafo conectado se mantenha ou aumente caso todos os nodos tenham um *degree* de 3 ou mais. Embora não seja regular, tem pelo menos 3 vizinhos e, tendo esta condição, foi tirada esta conclusão. Tendo em conta este mínimo, o número de utilizadores que o servidor fornece a cada novo *peer* é 3, caso seja possível, pois podem não estar utilizadores suficientes online.

O servidor deve aceitar que os *peers* peçam novos IPs para ligação, caso percam as suas ligações por alguma razão. Isto acontecerá raramente porque cada *peer* irá manter em memória os pares *ip:port* que conhece, ou seja, todos os que já passaram por ele. E tendo esta informação, sempre que precisa de novas ligações, poderá procurar nesses pares. Será fácil perceber, mais à frente, de que maneira estes pares *ip:port* passam por ele, mas um dos casos já foi falado: algum *peer* que o catálogo forneceu estava sobrecarregado e sugeriu um vizinho, ainda assim o *peer* guarda as suas informações porque podem ser úteis mais tarde. Um par é apagado quando se percebe ao tentar ligar que aquele IP já não está à escuta naquela porta.

Quando um *peer* se conecta a outro, cada um deles partilha informações sobre os utilizadores que subscreve, para iniciar uma troca de publicações relevantes, com base nas subscrições de cada um. Este processo chama-se *Peer Handshake*.

3.4 Propagação e obtenção de publicações

Quando um utilizador publica uma mensagem, esta será enviada para uma pequena percentagem dos nodos a que está ligado (se for possível, pode não estar ninguém online). Esta mensagem propaga-se pela rede, chegando potencialmente a alguns dos seguidores que a irão guardar. Para além de ser enviada apenas para uma percentagem dos vizinhos, existe um *time to leave* que, neste caso, não representa por quantos nodos vai passar, mas sim por quantos níveis. Por exemplo, um TTL = 2 faria com que uma mensagem fosse aos vizinhos da origem e aos vizinhos destes últimos. Este TTL não precisa de ser muito grande, pois não é necessário chegar a todos os subscritores, basta alguns para garantir que é possível obter a publicação, mesmo que a origem se desligue imediatamente a seguir.

Para um utilizador consultar a sua *timeline*, será feito *flooding* de um pedido com os seus subscritores e um id correspondente à última mensagem que viu de cada um deles. Se alguém tiver uma ou mais mensagens novas, envia para a origem do pedido (informação de ip e porta fornecida no pedido). Se a conexão efetuada entre os *peers* para envio de informação for nova e não violar os limites de nenhum dos participantes, mantém-se a conexão para reforçar a rede. O *flooding* tem um TTL do mesmo tipo que foi falado anteriormente. Neste caso, tendo em conta a experiência de Milgram sobre o problema do mundo pequeno [5], na qual se concluiu que existe uma separação de 6 "saltos"(em termos de conhecimentos) entre quaisquer 2 pessoas, o TTL neste caso será 6. Embora um grafo conectado gerado aleatoriamente, tipicamente, tenha um diâmetro pequeno devido à redundância de conexões existentes, não representa bem as conexões sociais que existiam na experiência de Milgram. Isto deve-se ao facto de, na realidade, quando um indivíduo A conhece B e B conhece C, significa que é provável que A conheça C. Este tipo de probabilidade não é representada num grafo gerado aleatoriamente, mas como neste caso as conexões para troca de informação (de interesse comum, pois é enviada informação que ambos subscrevem e um deles ainda não conhece) são aproveitadas para fortalecer a rede, acaba por ser representado, de certa forma, este comportamento social. Posto isto, tendo o TTL = 6 para o *flooding*, será coberta grande parte da rede com este pedido que apenas será propagado por 6 níveis.

3.5 Identificação de utilizadores e subscrições

Cada utilizador será identificado por uma *hash* do seu email. Optou-se pelo email para ter alguma garantia que será único, embora seja possível dois utilizadores colocarem o mesmo email. É impossível ter a certeza, porque um utilizador pode usar o email de outra pessoa. Também pode acontecer vários utilizarem emails falsos, tais como mail@mail.com. Assumimos que os utilizadores fornecem o seu email verdadeiro. A *hash* serve para facilitar a manipulação de ids de utilizadores, pois terá um tamanho previsível, contrariamente ao email.

A aplicação guarda informação sobre as subscrições do utilizador. É especificado pelo utilizador o email que pretende subscrever e é guardada a *hash* correspondente. Para cada uma das subscrições também é guardado o id da última mensagem conhecida. Isto torna possível pedir mensagens dos utilizadores que subscreve.

4 Implementação

4.1 Tecnologias

O servidor, designado como o catálogo que associa *peers* à sua localização na rede, foi implementado em Java, seguindo o paradigma da programação por eventos. Esta decisão teve consequência no maior conhecimento do paradigma escolhido na linguagem. Uma outra tecnologia escolhida foi o Python, como linguagem para a implementação da lógica dos *peers*, justificando-se a escolha como

sendo uma linguagem com um *throughput* de código por unidade de tempo maior que outras como Java, além de também ser uma boa escolha para a modelagem de protótipos.

5 Conclusões e Trabalho Futuro

O objetivo deste projeto era deliniar várias e estrategias adequadas de modo a ser implementada uma rede social *peer-to-peer*. O ponto de partida era a criação de um servidor/catálogo que armazena-se informação sobre os *peers*, seguido-se a implementação dos mesmos, o que acarretou vários desafios no que toca à propagação e obtenção de publicações bem como na identificação de utilizadores e subscrições.

Apesar de todas as decisões tomadas, alguns aspectos prevalecem ainda pouco afinados. Alguns exemplos passam por implementar um serviço tolerante a faltas, principalmente no que toca ao conhecimento do catálogo de *peers online*, assim como se na eventualidade de um *peer* falhar antes da escrita em disco do seu estado, este não manterá persistência a nível dos seus dados. Também o conhecimento de subscritores a um determinado utilizador pode ser incompleto quando o destino da subscrição estiver *offline*.

Um dos pontos fracos desta rede social é a autenticação e garantia de unicidade na identificação de utilizadores. Como trabalho futuro, considera-se a possibilidade de utilizar um protocolo de autenticação *peer to peer*, o PAuth [6]. Neste protocolo existem chaves pública e privada para todos os utilizadores e sempre que se tentam ligar inicia o protocolo. Quem se pretende autenticar envia a sua chave pública para um servidor central que escolhe k *peers* e envia os seus IPs para o utilizador. De seguida o utilizador contacta cada um deles e envia-lhes a sua chave pública para iniciar uma prova de posse de chave privada. Cada *peer* gera uma string random, cifra com a chave pública e envia o resultado para o utilizador a autenticar. O utilizador responde com a informação decifrada. Os *peers* verificam a resposta e enviam para o servidor central informação sobre a prova. Se o servidor receber um determinado número de confirmações, autentica o utilizador.

Referências

1. Evan Prodromou, Matt Lee, Mikael Nordfeldth *https://gnu.io/social/*
2. Diaspora* *https://diasporafoundation.org/about*
3. Federico Marani. *https://github.com/fmarani/newton*
4. Nicholas C. Wormald. *The asymptotic connectivity of labelled regular graphs.* Journal of Combinatorial Theory, Series B, 31(2):156–167, 1981.
5. S.Milgram. *The small world problem.* Psychology Today 1(1967).
6. Zijing Gao, Thomas Lu, Anand Srinivasan. *PAuth: A Peer-to-peer Authentication Protocol.* March 20, 2015.