Universidade do Minho

Mestrado Integrado em Engenharia Informática

Departamento de Informática

Sistemas Distribuídos em Larga Escala

Decentralized Timeline

Diogo Figueiredo Pimenta A75369
Isabel Sofia da Costa Pereira A76550
Maria de La Salete Dias Teixeira A75281

19 de Maio de 2019

Conteúdo

1 Introdução

O presente relatório descreve a arquitetura desenhada, as decisões tomadas e os problemas encontrados no desenvolvimento de um serviço de cronologia (como por exemplo o Twitter) descentralizado, no âmbito da UC de Sistemas Distribuídos em Larga Escala. Neste serviço, cada utilizador deve ser capaz de criar *posts* e de subscrever a *posts* de outros utilizadores, armazenando esse conteúdo, tornando possível passá-lo a outros caso o criador original se encontre *offline*.

Dada a familiaridade dos elementos do grupo com programação concorrente e de sistemas distribuídos, em grande parte devida a UCs anteriores, o grupo decidiu por unanimidade implementar este serviço na linguagem **Java** com auxilio das componentes de comunicação assíncrona da biblioteca **Atomix**.

2 Arquitetura Global

Os requisitos impostos a este projeto já implicam, por si, uma arquitetura de uma determinada natureza. Sendo então este serviço descentralizado, ou *peer-to-peer*, é diretamente necessário que o principal componente do serviço seja o próprio **cliente**. No entanto, este componente por si só não é suficiente. Numa situação em que um dado cliente é novo, ou em que o grafo que representa a rede não se encontra totalmente ligado, surge a necessidade de existir uma entidade fixa conhecida por todos que, quando necessário, dá um "empurrão" na rede. Esta entidade será posteriormente referida como servidor de ***bootstrap***.

3 Arquitetura de um *Peer*

3.1 Cliente

Focando na arquitetura local do cliente, este terá de ser obrigatoriamente ***multi-threaded***. Esta necessidade prende-se no facto de cada cliente ter de ser capaz de estar a interagir com o utilizador e, ao mesmo tempo, estar a realizar comunicações assíncronas com outros clientes, de modo a conseguir mostrar informações atualizadas ao cliente sem bloquear a aplicação, caso isso seja permitido pela configuração da rede no momento.

3.1.1 Classe *node*

Para minimizar a complexidade da implementação do *peer* e a possibilidade de erros, toda a informação necessária para o funcionamento de um cliente tem como ponto de entrada a mesma classe, *node*, onde é forçado encapsulamento total e controlo de concorrência através do uso de ***synchronized***.

As **informações necessárias** passam por este possuir um *IP*, um identificador único, estruturas para armazenamento dos seus *posts*, dos *posts* de quem subscreve e das sugestões destes e, por fim, informação para contactar os seus vizinhos e os utilizadores a quem subscreve.

Todas as operações contidas nos métodos desta classe ou são de complexidade reduzida e tempo de execução quase instantâneo, ou cingem-se a criar uma cópia profunda dos dados necessários e a devolvê-los para a *thread* os poder processar sem bloquear todo o programa.

3.1.2 Thread responsável pela interação com o utilizador

A *thread* de interação com o cliente tem, portanto, um funcionamento simples. Limita-se a expor opções ao cliente, a aplicar *gets* e *sets* de acordo com a opção escolhida e, finalmente, a mostrar ao cliente a informação obtida. Essas opções são as seguintes:

- Publicar um *post*;
- Consultar o seu perfil, isto é, todos os *posts* por ele publicados, por ordem causal;
- Consultar todos os *posts* publicados por quem subscreve, por ordem causal e cronológica;
- Consultar todos os *posts* publicados por um utilizador em particular a que subscreve, por ordem causal;
- Subscrever a um utilizador que age como seu vizinho;
- Subscrever a um utilizador dando a chave pública deste;
- Consultar as sugestões de utilizadores dadas pelos clientes que subscreve, isto é, quem esses utilizadores seguem;
- Cancelar subscrição a um utilizador a que esteja subscrito.

É, portanto, na *thread* de comunicação assíncrona que se encontra a essência do serviço, pois é esta a responsável por adicionar os dados atualizados/novos ao *node*.

3.1.3 Thread responsável pela comunicação assíncrona periódica

Primeira comunicação

Para melhor expor o processo de comunicação com o exterior por parte de cada *peer*, comecemos pela primeira situação em que ele é inserido: iniciado, sem estado prévio a carregar e sem conhecimento de outros *peers* na rede. Neste caso, existe apenas uma ação possível para o cliente, contactar o servidor de *bootstrap* para obter conhecimento sobre os clientes existentes na rede, assumindo, 5 desses, o papel de seus vizinhos. Caso o cliente inicie sessão com um estado anterior, apenas o seu *IP* pode ser atualizado, sendo enviado para o *bootstrap* a fim de o atualizar também. Neste caso, o *bootstrap* volta a enviar as informações dos clientes da rede, para o caso de estas terem sido atualizadas enquanto o utilizador em questão se encontrava *offline*.

Comunicação periódica

Tendo conhecimento de outros *peers* na rede, o cliente vai iniciar o seu processo de *refresh* periódico. Em cada período de *refresh* vai tentar contactar os utilizadores que subscreve para obter as suas sugestões de utilizadores e os seus *posts*, estando estes segundos acompanhados de um identificador causal, que indica a partir de que *post* o cliente ainda não recebeu.

Como estes podem se encontrar *offline*, no caso dos *posts*, é marcado que se tentou contactar esse *peer*. O mesmo não se aplica às sugestões pois não é uma funcionalidade crucial para o funcionamento do sistema, sendo que se os mesmos não responderem, não se considera ser um problema.

Sendo assim, iremos continuar a ver o problema do lado dos *posts*. Quando recebe a resposta de cada um desses *peers* ao pedido referido, marca-se o contacto com ele como finalizado. Caso no período seguinte algum desses *peers* ainda não tenha respondido, o conteúdo desse *peer* é pedido a outros, nomeadamente os seus vizinhos. Caso esses também não consigam satisfazer o pedido, também o redirecionam para os seus vizinhos. Este processo pode, portanto, inundar rapidamente a rede e nunca terminar. Para evitar este final drástico, quando um conteúdo é pedido a terceiros, o pedido é etiquetado com um TTL, semelhante ao encontrado no protocolo *TCP/IP*. Cada *peer*, antes de reencaminhar o pedido de conteúdo, decrementa o TTL incluído no pedido. Chegando ao valor 0, o pedido deixa de ser redirecionado. No pior dos casos, caso não consiga receber nenhuma resposta a um pedido de *post*, isto é, se grande parte da rede se encontrar *offline*, o *bootstrap* é contactado novamente para este providenciar mais vizinhos ao *peer* em questão.

3.2 *Bootstrap*

O servidor *bootstrap* está encarregue de armazenar a informação básica de todos os *peers*, isto é, o identificador (chave pública) e o endereço mais recente destes. A relevância desta informação deve-se ao facto de este ter de, tal como foi mencionado anteriormente, fornecer a um utilizador novos vizinhos, quando necessário, e a rede toda atualizada, quando este se liga. Sendo assim, é um pressuposto de que o servidor de *bootsrap* se encontra num endereço fixo, conhecido pelos clientes, e que se encontra sempre ligado. Estes fatores levam a que a comunicação *peer-bootstrap* siga o padrão clássico de *request-reply*, suportando assincronia, pois a ligação pode ser lenta.

No entanto, esta entidade apresenta um problema devido ao facto de devolver a rede toda a um utilizador, o que afeta o sistema em termos de desempenho caso a rede contenha muitos *peers*. Apesar da consciência do grupo sobre este tópico, esta implementação foi apenas para simplificar o problema, sendo que uma possível solução seria o utilizador pedir a rede atualizada a partir de certo momento do tempo, isto é, a partir do momento em que se desligou e deixou de receber atualização de endereços. Este procedimento aliviaria a carga que tal mensagem causa ao *bootstrap*, permitindo que a escalabilidade do sistema não

fosse afetada.

4 Atualização dos endereços entre *peers*

Devido ao facto de os utilizadores poderem ficar *offline*, os seus endereços, ao iniciar novamente sessão, podem se modificar. Isto significa que quando um nó pretende enviar uma mensagem a um determinado utilizador que tenha iniciado sessão depois de si, tal pode não ser possível. Esta situação acontece porque o primeiro possui o endereço antigo do segundo (caso tenha de facto sido alterado), provindo da rede enviada pelo *bootstrap* quando este efetuou o *login*, tal como referido anteriormente. Para resolver esta situação, quando um utilizador entra novamente na rede, este envia para todos os outros uma mensagem a anunciar as suas informações, isto é, o seu identificador, *username* e endereço atualizado. Com isto, é possível então que um utilizador que já se encontre com sessão iniciada, consiga obter o endereço mais recente de um outro que iniciou sessão posteriormente, atualizando assim a informação que tem dos utilizadores da rede, permitindo que qualquer nó *online* possa ser contactado por outro.

5 Causalidade nos *posts*

Como a ordem a que o utilizador lê as mensagens é importante, cada *post* é marcado com um identificador, desta forma, o utilizador nunca lê um *post* sem ler o anterior. Os *posts* que ainda não podem ser apresentados, isto é, tem um identificador maior ao qual se está à espera, são guardadas numa lista de espera. Essa lista é verificada sempre que se aceita *posts* novos.

6 Causalidade nas mensagens

A causalidade nas mensagens foi apenas implementada na mensagem que o cliente envia ao *bootstrap* para este atualizar o endereço do utilizador em questão. Isto deve-se ao facto de considerarmos que os *IPs* nessa entidade têm de ser sempre os mais recentes, para que, quando o *bootstrap* der a informação desse *peer* a outro, o endereço estará acessível para receber pedidos, caso o utilizador ainda se encontre *online*.

No entanto, existem outras três mensagens onde os endereços também apresentam um papel importante, nomeadamente, a mensagem que um utilizador envia a todos os outros que se encontram *online* para atualizarem o seu endereço e as mensagens que o *bootstrap* envia ao utilizador, isto é, a mensagem sobre a rede e a mensagem com novos vizinhos. Apesar de a ordem nestas ser relevante, tal apenas acontece se o utilizador iniciar sessão, instantaneamente se desligar e voltar a ligar-se, causando que a mensagem 1 possa ser recebida depois da mensagem 2, atualizando a rede para endereços antigos. Desta forma, a sua implementação não foi concluída por considerarmos um problema leve, no entanto, será futuramente.

Nas restantes mensagens, isto é, nas mensagens de *post* e *suggestions*, não se verifica essa propriedade devido ao facto de considerarmos que a sua ordem não é relevante neste sistema. Por exemplo, no pedido de um *post*, o pedido 1 ou 2 seria igual, sendo que a resposta ao mesmo também não afeta qual se recebe primeiro porque os *posts* em si são aceites pelo seu próprio identificador causal. A única mensagem dessas em que a ordem poderia ser relevante seria ao receber as sugestões de um *peer*, porque a resposta 2 contém informação mais atualizada que a 1, no entanto não consideramos ser uma funcionalidade crucial, sendo que se o utilizador visualizar a lista antiga de sugestões, não é um problema.

7 Identidade do cliente

O facto de o serviço ser descentralizado faz também surgir um problema fundamental na arquitetura dos *peers*. Como identificar os utilizadores? Dada a inexistência de uma entidade central, como no caso do Twitter, nada impede um utilizador de colocar nas mensagens o nome de outro, fazendo-se assim passar por ele. Como possível solução, o grupo considerou assinar digitalmente cada *post*, ou seja, cada utilizador ao criar um *post*, encripta uma *hash* do texto escrito com a sua chave privada e anexa a sua chave pública e essa *hash* encriptada com o *post*. Deste modo, é possível a qualquer utilizador dois dados *posts* pertencem realmente ao mesmo autor. No entanto, como este pormenor começa a "fugir"um pouco ao âmbito da UC, o grupo decidiu simplificar este passo e simplesmente anexar a chave pública aos *posts*, servindo também como autenticação do utilizador no sistema.

8 Recuperação do estado

O não-repúdio da autoria dos *posts* não foi a única funcionalidade simplificada devido ao âmbito da UC. Como requisito, o utilizador deve ser capaz de ficar *offline* e de voltar ao estado em que se encontrava. De forma a simplificar a implementação e ultrapassar a necessidade de criar uma base de dados para garantir persistência dos dados, os objetos relevantes (*node* e *bootstrap*) a serem guardados implementam a interface *Serializable* e são escritos em ficheiro sempre que sofrem alguma alteração nos seus dados, podendo assim recuperar-se o estado anterior no próximo inicio de sessão, podendo mudar apenas o *IP* do cliente. Este passo foi dado com consciência, por parte do grupo, de que esta funcionalidade é muito pouco eficiente quer em escrita como leitura, e que, caso a aplicação fosse usada no "mundo real"esta forma de persistência não seria viável. Além disso, como se pode acabar por guardar bastantes *posts* de utilizadores a que se subscreve, a cada inicio de sessão, esses *posts* são eliminados caso possuam mais de uma semana de existência.

9 Conclusão

A realização deste trabalho mostrou-se ser um desafio, quer a nível de desenho da solução quer a nível de implementação, especialmente devido à natureza assíncrona do programa.

No entanto, a solução desenvolvida foi testada com sucesso em todas as funcionalidades propostas e desenvolvidas. Esses testes são de natureza local, em que os endereços dos clientes e *bootstrap* apenas diferem na porta utilizada.

Supõe-se ainda que a arquitetura e a solução desenvolvida têm a capacidade de escalar para um número elevado de nós, mas não para um número demasiado elevado, dada a limitação no modo de manter os *peers* completamente ligados. No entanto, não foi possível testar completamente o nível de escalabilidade por limitações de *hardware* para criar um número elevado de nós sem afetar a *performance* do servidor de *bootstrap*.

Caso o desenvolvimento do projeto fosse a ser continuado, o grupo considera que seria necessário testar e melhorar a robustez do programa, para um número elevado de nós, e implementar a garantia de não-repúdio da autoria dos posts, ou seja, possibilitar a prova de que dois dados *posts* pertencem ao mesmo autor. Como prioridade na continuação do desenvolvimento, o passo mais importante será a implementação da noção de vistas, de modo a reduzir o tamanho das mensagens trocadas entre os *peers* e o *bootstrap* cada vez que um *peer* se liga, e a implementação da causalidade nas mensagens onde o endereço representa um papel importante.