Escola de Engenharia

Universidade do Minho

PROJETO DE SISTEMAS DISTRIBUIDOS

2020/2021

Realizado por:

JOÃO TORRES – A85846

JOSÉ PIRES – A87980

JOSÉ MANSO – A87961

RITA GOMES – A87960

Conteúdos

Introdução

O presente relatório tem como objetivo descrever o processo de construção de uma plataforma inspirada no problema do rastreio de contactos e deteção de concentração de pessoas devido à COVID-19. Para proceder à resolução desta aplicação iremos abordá-la sob a forma de cliente/servidor em Java utilizando sockets e threads.

Primeiramente, serão apresentadas as classes e as respetivas caracterizações. Posteriormente, expomos o raciocínio seguido para a construção das funcionalidades que o serviço deverá suportar. Finalizamos o relatório com uma breve conclusão que, após uma pequena exposição das dificuldades sentidas no desenvolvimento do projeto, pretende retratar a nossa perceção geral do trabalho realizado.

Estruturação do projeto

A aplicação deve estar organizada de modo a que todos os utilizadores consigam estabelecer uma conexão segura e estável com o servidor e que este, por sua vez, tenha a capacidade de gerir os dados relativos aos utilizadores, recorrendo muitas vezes ao auxílio de Controllers, isto é, classes que se responsabilizam por toda a lógica do sistema.

Desta forma, definimos a classe User que será constituído pelo seu nome de usuário, password, localização (composta por um par (x, y), visto que a grelha em que a aplicação atua é definida em N linhas por N colunas), e por uma verificação da existência de uma autorização especial, que lhe poderá permitir acessibilidade a novas funcionalidades. Para além disso, a classe Server irá conter duas variáveis do tipo DataInputStream e DataOutputStream, respetivamente, para que consigam receber a informação dos pedidos de cada utilizador e a consigam transmitir às classes responsáveis. De seguida, de modo a armazenar os dados de todos os utilizadores, criámos a classe UsersController, que contém apenas um Map em que a chave é o nome de usuário e o valor corresponde ao utilizador em questão.

Isto seria suficiente caso apenas pretendêssemos saber, por exemplo, quais utilizadores usam a aplicação ou quantos utilizadores se situam numa determinada localização. No entanto, o facto de existir apenas um Controller limita o número de funcionalidades que a aplicação pode fornecer. Por isso, decidimos criar a classe RegisterUsers, que tem o objetivo de guardar o histórico de utilizadores que já estiveram no mesmo local de cada utilizador. Por exemplo, caso um utilizador A se tivesse encontrado com um utilizador B e um outro utilizador C ainda não tivesse estado em contacto com nenhum utilizador, a o histórico de cada um será, respetivamente: {B}, {A}, {}. Este Controller será útil sempre que algum utilizador informar a aplicação que está infetado, pois desta forma é possível ter acesso à sua lista de utilizadores e informá-los do risco de infeção.

Com a criação de vários Controllers, consideramos necessário que existisse uma outra classe responsável por decidir, para cada pedido de um utilizador, qual dos Controllers seria o mais indicado para o satisfazer. Daí, surgiu a classe ControllerSkeleton que implementa a interface Skeleton, com o método *handle*. Este método abrange o conjunto de pedidos que um utilizador (regular ou premium, caso tenha a autorização especial) pode fazer e, com base nesse mesmo pedido, decide qual (ou quais) dos Controllers é que deve retirar os dados pretendidos.

Para que a classe User, para além de armazenar a informação de um utilizador, não ficasse responsável por estabelecer uma conexão com o servidor, a classe Stub irá precisamente encarregar-se de tal tarefa, sendo constituída por uma variável do tipo DataOutputStream (envia pedidos ao servidor, juntamente com as informações necessárias) e outra variável do tipo DataInputStream (recebe os resultados dos pedidos anteriormente realizados).

A classe User, assim como todos os Controllers até ao momento criados, dispõem dum ReentrantReadWriteLock, que é dividido por um lock de escrita e um lock de leitura. Assim, a aplicação pode funcionar de modo a que todas as operações que apenas requeiram leitura sejam feitas concorrentemente sem haver filas de espera. No caso das operações de escrita, não é possível que o mesmo possa acontecer. Ainda assim, como os utilizadores apenas têm a capacidade de gerir as suas próprias informações, o único caso que pode levantar alguns problemas é no processo de registo de um utilizador, pois é necessário inserir um User no Map do UsersController usando um lock de escrita.

Funcionalidades

Obviamente, nem todas as funcionalidades (ou *queries*) funcionam da mesma forma, mas de forma generalizada e de acordo com a estrutura implementada, uma *query* segue este tipo de procedimento:

1. Utilizador envia um pedido ao respetivo Stub
2. Stub envia o pedido ao Server, utilizando a variável do tipo DataOutputStream
3. Server recebe o pedido, através da variável do tipo DataInputStream
4. Server envia todos os dados necessários à resolução do pedido para o ControllerSkeleton
5. ControllerSkeleton decide qual (ou quais) dos Controllers usar para responder à *query*
6. É realizado o método necessário dentro do Controller
7. Controller retorna o resultado do método ao ControllerSkeleton
8. Controller retorna o mesmo resultado ao Server
9. Server retorna o resultado ao Stub, utilizando a sua variável do tipo DataOutputStream
10. Stub recebe o resultado proveniente do Server, utilizando a variável do tipo DataInputStream
11. Stub envia o resultado ao utilizador respetivo

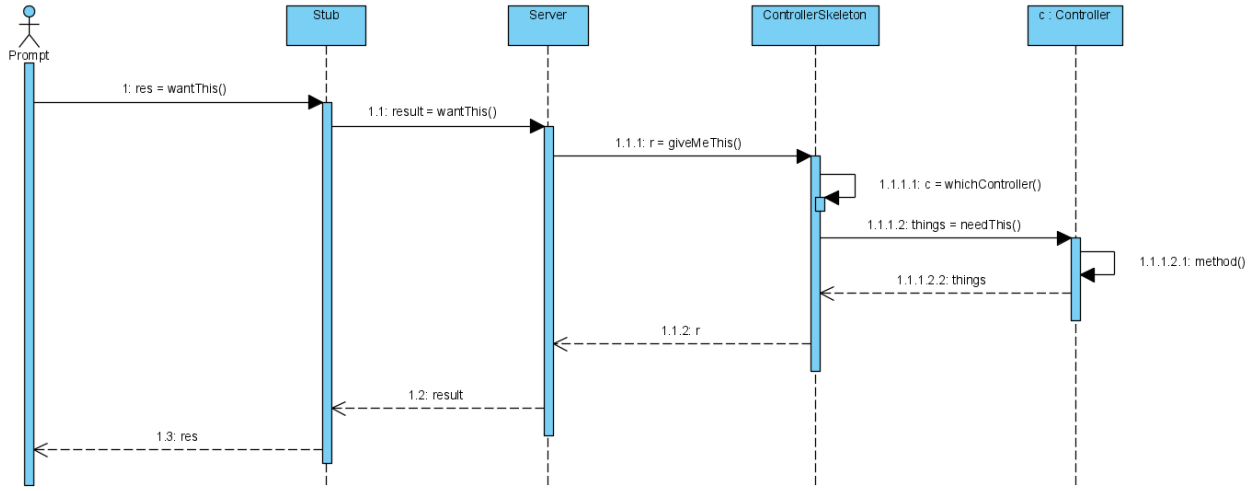

Figura 1 - Funcionamento de uma query.

Foi proposto um conjunto de funcionalidades, às quais a nossa aplicação suporta da seguinte forma:

- Registo e autenticação de um utilizador: previamente, é feita uma conexão entre o utilizador e o servidor. De seguida, procede-se ao registo, enviando o username, password e verificação da autorização especial. Caso o username inserido já exista, o registo não é validado e o utilizador terá de tentar novamente com outro nome. Caso contrário, o utilizador pode-se autenticar, introduzindo o seu nome de usuário e password. O processo de autenticação falha se a password não corresponder à que foi introduzida no registo, anteriormente.

- Manter o servidor informado quanto à localização do utilizador: o utilizador deverá, manualmente, avisar o servidor sempre que mude de localização. Esse processo de alteração é bastante simples, em que apenas a variável correspondente à localização é alterada, usando um lock de escrita. Para além disso, existe a possibilidade de o utilizador se contactar com novos utilizadores ao mudar de local, por isso o Map de registos de contactos presente no RegisterUsers é atualizado devidamente.

- Saber o número de pessoas numa determinada localização: o utilizador pretende deslocar-se para um local, e por isso pretende saber quantas pessoas se encontram lá. Esse método passa por fazer uma travessia pelo Map de utilizadores presente no UsersController e contar o número de utilizadores que contêm a localização em questão.

- Saber quando não há ninguém numa dada localização, com o intuito de se deslocar para lá: o utilizador começa por comunicar a sua intenção de se deslocar a um determinado local. A partir desse momento, é criada uma thread exclusivamente com o propósito de enviar, assim que possível, a mensagem de que a localização que o utilizador se pretende deslocar está vazia. Note-se que a thread que usa o pedido de comunicação apenas é usada de 5 em 5 segundos para que não congestione o servidor, utilizando um método *timeout* que adormece a thread durante um certo intervalo de tempo.

- Comunicar infeção: o utilizador comunica que está infetado, bloqueando assim todas as funcionalidades que anteriormente tinha acesso. Assim que é comunicada a sua infeção, realiza-se o *logout*, não permitindo que este utilizador se volte a autenticar.

- Notificar pessoas potencialmente contagiadas: sempre que um utilizador A comunica a sua infeção, é possível obter a lista dos utilizadores que A teve contacto (isto é, na mesma localização). Através dessa lista, realiza-se uma travessia em que todos os utilizadores são avisados com uma mensagem que evidencia o risco de contágio de alguém que esteve perto deles. Para cada utilizador potencialmente contagiado, é criada uma thread à parte com um *timeout* de 10 segundos que irá fazer o pedido de verificação ao servidor. No entanto, existem dois casos a ter em conta:

    1. O utilizador potencialmente contagiado está conectado e autenticado com a aplicação e recebe a notificação de um outro utilizador infetado.
    2. O utilizador potencialmente contagiado não está autenticado, e por isso apenas quando fizer *login* irá receber a notificação. Para isso, limitamos o número máximo de notificações a 1, pois durante o tempo em que o utilizador esteve desconectado da aplicação, vários utilizadores poderão ter comunicado a sua infeção. Por isso, de modo a simplificar o número de mensagens que o utilizador potencialmente contagiado pode receber, decidimos que apenas irá receber uma mensagem assim que se autenticar, referindo que ele esteve em contacto com um ou mais infetados.

- Carregar mapa de localizações: esta funcionalidade apenas pode ser acedida por utilizadores que têm uma autorização especial. Através de uma constante que delimita o tamanho da grelha, o ControllerSkeleton seleciona o UsersController para criar um mapa em que, para cada localização, são listados os utilizadores que lá se encontram de momento. Caso não haja ninguém, a lista será apresentada como vazia. Note-se que a constante que delimita o mapa tem o valor 5. Por isso, as localizações válidas encontram-se entre (0,0) e (4,4).

Discussão e conclusão

Com a realização deste trabalho foi possível consolidar e colocar em prática os conhecimentos obtidos ao longo da unidade curricular de Sistemas Distribuídos. Para além disso, adquirimos um maior domínio da linguagem Java, destreza no uso da mesma e também uma maior espontaneidade no processo de tomar decisões lógicas para resolver problemas no contexto de programação com sockets e threads.

Tal como sabemos, o objetivo dos Sistemas Distribuídos é otimizar o desempenho de maneira a que seja possível dividir tarefas e processá-las separadamente, criando paralelismo e, consequentemente, diminuir o tempo de execução.

No processo de desenvolvimento do projeto, notamos algumas dificuldades nomeadamente na criação de threads e na maneira como um utilizador podia obter várias. Para além disso, notámos alguns casos de corridas devido a alguma dificuldade na utilização dos locks.

Concluímos ter feito um bom trabalho face àquilo que nos foi proposto e consideramos que os conhecimentos adquiridos serão úteis em projetos futuros. No final de contas, a aplicação está preparada para suportar problemas como autenticações inválidas e alterações de localização não aceitáveis (por estarem fora do limite da grelha).