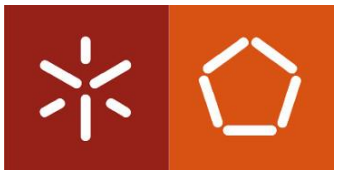

UNIVERSIDADE DO MINHO

DEPARTAMENTO DE INFORMÁTICA

TRABALHO PRÁTICO

CovidApp

Sistemas Distribuídos
24 de janeiro de 2021

(a) Ana Catarina Canelas
A93872

(b) Ana Filipa Pereira
A89589

(c) Carolina Santejo
A89500

(d) Raquel Costa
A89464

Índice

Índice de Figuras

Introdução

Neste trabalho da UC de Sistemas Distribuídos foi pedido que criássemos um programa que executasse em ambiente *multi-thread,* semelhante á aplicação *Stay Away Covid.* Tendo em conta o contexto atual, com o número crescente de casos diários, é fundamental controlar o número de contágios. Para tal, a existência de uma aplicação de rastreio da localização das pessoas é essencial, para facilitar a identificação de contactos próximos, bem como evitar aglomerados. Neste programa, os utilizadores autenticam-se e têm várias opções disponíveis tais como saber quando uma localização se encontra vazia, quantas pessoas se encontram num dado local, registar uma infeção ou deslocar-se para uma dada posição. Foi necessário, também, considerar a existência de dois tipos de *users*, os normais e os especiais, sendo que estes últimos têm a funcionalidade extra de descarregar o mapa com informações de infetados e de visitantes dos vários locais.
Para realizar este trabalho, foi necessário aplicar diversos conceitos aprendidos nas aulas, nomeadamente o controlo de concorrência (primitivas lock/unlock), variáveis de condição, e a arquitetura cliente-servidor.

Arquitetura

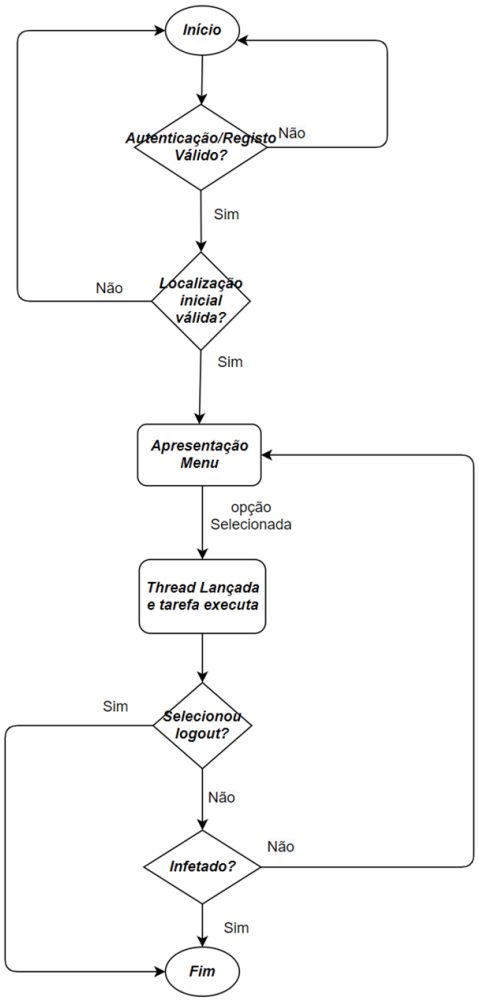

Figura 1 - Diagrama Geral

Servidor

O Servidor inicializa um *socket*, conhecido a todos os clientes, no port "12345", onde será possível a troca de mensagens com o Cliente. Assim que é estabelecida a conexão, qualquer *request* feito por um cliente poderá ser respondido pelo Servidor, e para tal, ele irá aceder à classe *GestInfo* que contém todas as classes presentes no package "GestInfo", e tem ainda todos os dados necessários armazenados em memória. Com base nisto, irá conseguir satisfazer qualquer pedido feito pelo cliente.

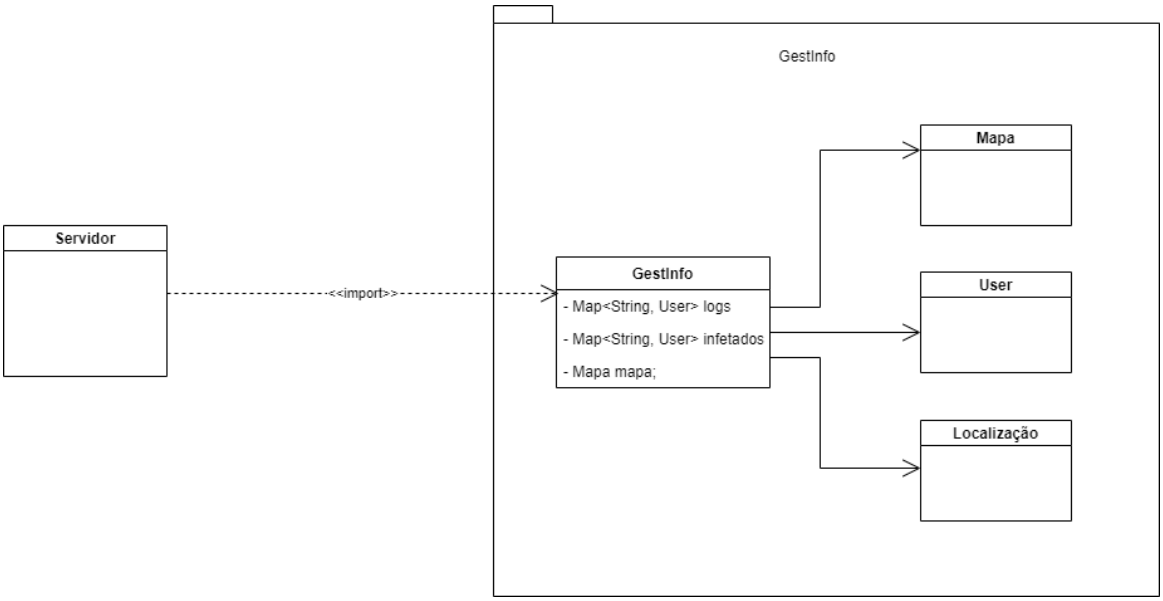

Figura 2 - Diagrama de Classes

Cliente

Neste programa, o cliente conecta-se ao servidor através do socket criado (com o port "12345"). Após a conexão com o servidor, um menu é apresentado com várias opções. A seleção de cada uma das opções leva á criação de um a thread de execução. O cliente, no software, apenas se preocupa em enviar pedidos ao server, e receber as suas respostas.

Quando o cliente se conecta, este tem duas opções: efetuar registo ou fazer login. O registo de um user só é válido se não existir alguém com o mesmo username, enquanto que a autenticação só é válida se o username e a passoword correspondente já estiverem guardadas no sistema. Após o registo ou login ser efetuado com sucesso é apresentado ao cliente um outro menu mais complexo no qual estão propostas todas as funcionalidades do programa.

Além disto, entre todas as opções do menu, as únicas que encerram a comunicação com o servidor é o logout e o comunicar infeção.

Comunicação Cliente ↔ Servidor

Para que um cliente possa comunicar com o servidor e vice-versa foi necessário utilizar um socket que estabelece a conexão e definir duas classes essenciais que são responsáveis por encaminhar os dados no socket: Demultiplexer e TaggedConnection.
Para compreender melhor como este procedimento é realizado, é importante distinguir as duas direções de comunicação: Cliente → Servidor e Servidor → Cliente.

- **Cliente → Servidor**

Quando um cliente quer enviar dados para o servidor invoca o método send da classe *Demultiplexer*, que por sua vez chama também o metodo *send* da classe *TaggedConnection*. Estes dois métodos vão encaminhar os dados no socket junto com a sua *tag*, sendo que esta serve para que o servidor possa distinguir o tipo de mensagem que é enviada. Por exemplo, no caso deste projeto cada valor corresponde a uma funcionalidade.
Para o servidor receber os dados apenas invoca o método *receive* da classe TaggedConection que retorna um objeto da classe frame que inclui a *tag* e os dados enviados previamente pelo cliente.

- **Servidor → Cliente**

A direção Servidor → Cliente implicou um maior cuidado uma vez que o cliente implementado é *multi-threaded* e é necessário que as respostas do servidor sejam enviadas para as threads corretas. Para enviar dados o servidor apenas invoca o método *send* da classe *TaggedConnection,* que, como já referido anteriormente escreve no *socket* a *tag* e os dados da mensagem.

De modo a organizar as respostas recebidas no cliente, no início da sua execução, é invocado o método start da classe *Demultiplexer,* que vai executar uma *thread* responsável por "intercetar" todas as mensagens enviadas pelo servidor para o socket e organizá-las num *buffer* que junta todos os dados recebidos de acordo com a sua *tag* numa fila de espera. Assim, quando uma thread do cliente quer receber os dados basta invocar o método receive da classe *Demultiplexer* que vai procurar os dados correspondentes na fila de espera do buffer com a sua *tag* e retorná-los. Se a fila estiver vazia, a *thread* vai ficar à espera que cheguem dados (método await).

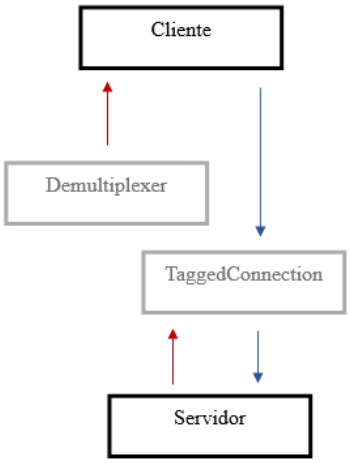

Figura 3 - Comunicação Cliente < - > Servidor

Funcionalidades

Através do diagrama da figura abaixo, é possível perceber quais são os menus que são apresentados ao longo da execução, bem como as diversas funcionalidades que o programa possui (e que várias delas já foram referidas neste relatório).

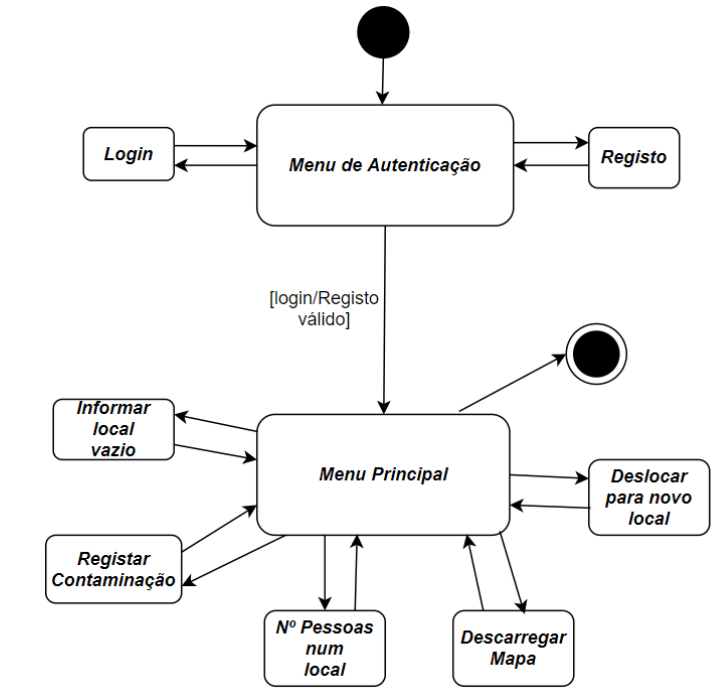

Figura 4 - Diagrama de Estados

Autenticação e registo

Logo que um cliente se conecta ao socket do servidor, é requerido o seu registo (opção 1) ou autenticação/login (opção 2), caso se encontre registado. Em ambos os casos, são solicitados um nome de utilizador e a sua palavra passe, sendo que no registo é ainda pedida a confirmação de utilizador com autorização especial. Este utilizador, além das funções habituais, terá ainda acesso a uma funcionalidade adicional (descarregar mapa) que será descrita mais abaixo. Assim, serão enviados todos os campos necessários em formato binário para o socket com a tag correspondente (1 ou 2).

No caso do servidor vai ser realizada a conversão dos dados recebidos para os tipos correspondentes. Para o login, é feita a correspondência com os valores presentes nos logs guardados em memória e são enviadas exceções caso não se verifique uma delas (nome ou palavra passe).

Para efetuar o registo, é adicionado um utilizador ao Map dos logs desde que não esteja previamente registado.

De modo a saber todas as conexões que vão sendo efetuadas com o servidor é guardado em memória um *Map* com todas as *TaggedConection* criadas, associadas ao nome do utilizador que está a efetuar a respetiva conexão. No final do registo ou do login é adicionada uma nova *TaggedConection* a este Map.

Por fim, o servidor envia a mensagem de confirmação para o socket e, no caso da funcionalidade ter sido concluída com sucesso, é mostrada a nova interface ao utilizador com as opções às quais tem acesso no programa.

Atualizar localização

Nesta funcionalidade, o cliente indica as coordenadas da localização para a qual ele pretende deslocar-se. Por sua vez, esta mensagem é enviada do cliente para o servidor. Assim que o servidor a recebe, em conjunto com a classe GestInfo, que armazena todos os dados em memória necessários para o funcionamento da aplicação, irá atualizar as coordenadas do correspondente objeto "User" do cliente, e irá atualizar o mapa, isto é, aumentar o número de pessoas da localização escolhida e diminuir o número de pessoas da localização anterior. Tendo sempre em mente que para fazer estas atualizações é necessário o uso de *locks/unlocks*. Além disso, sempre que um Cliente/User faz uma deslocação, a nova localização é adicionada ao seu histórico, com o respetivo instante de tempo em que ele se deslocou.

Número de pessoas numa dada localização

Para o tratamento desta funcionalidade o cliente tem de fornecer as coordenadas (x,y) da localização para a qual ele quer obter informação.
É enviada uma mensagem com estas coordenadas ao servidor, onde este irá processar os dados e com acesso à classe GestInfo, irá obter a resposta correta para o "request" feito pelo cliente. Assim que este obtém o número de pessoas da localização dada envia a resposta ao cliente que fez o pedido.

Notificação de uma dada localização vazia

O Cliente terá de indicar quais as coordenadas (x,y) da localização para a qual ele pretende receber uma notificação de quando esta se encontrar vazia, com a intenção de poder deslocar-se para a mesma assim que quiser. É importante referir que esta notificação é assíncrona, isto é o cliente tem acesso a outras funcionalidades e poderá executá-las enquanto espera pela notificação da localização vazia.

Primeiramente, além da thread que é lançada na classe **Cliente** quando o cliente escolhe esta funcionalidade, é também lançada uma outra thread que fica à espera de receber uma mensagem do **Servidor** para indicar que o local selecionado pelo cliente se encontra vazio. Como não usamos o comando *join()* para a thread , é possível correr outras threads simultaneamente.
Mas antes disto, é fulcral enviar ao servidor a localização. Este irá verificar se as coordenadas estão corretas e se o cliente não se encontra nas coordenadas que indicou, caso ambas as condições sejam verdadeiras, então este irá executar uma thread que invoca o método *getLocalVazio()* da classe Mapa. Neste método é feito um lock ao local ao verificarmos o número de pessoas que se encontra no local, de forma a que nenhuma outra thread interfira no processo e altere a informação que nós queremos obter. Caso este número seja maior então entramos num ciclo que irá esperar por um signal do método retirar() da classe Localização. Como é pouco eficiente estar constantemente a verificar o número de pessoas numa localização, o grupo optou pela estratégia de verificar o número de pessoas apenas quando fossem retiradas pessoas da localização, e após isso, caso o número ficasse a 0 era então mandado um signalAll() para a thread ou todas as threads à espera pela Condition cond. Voltando, portanto, ao método getLocalVazio(), assim que recebe o signal, o número de pessoas na localização dada é 0 e portanto ele sai do ciclo e é lançada uma

exceção. Acontecendo isto, a thread a ser executada no Servidor é interrompida e o Servidor envia uma mensagem ao Cliente a notificar a Localização Vazia.

Comunicação de contaminação

Quando um utilizador regista no sistema que está contaminado então automaticamente é considerado que este encontra-se em isolamento, deixando de existir no nosso programa. Portanto fechamos a conexão entre esse cliente e o servidor, isto é, o *socket*. Além disso, o objeto User correspondente a este cliente é retirado da estrutura de dados "logs" da classe *GestInfo* e a flag que distingue se um user é infetado ou não é colocada a 1, após isto, o objeto é adicionado à estrutura "infetados" de modo a termos acesso ao histórico desse cliente para rastrear os seus movimentos e contactos com outros clientes. Sendo assim, este cliente não poderá voltar a fazer login no programa porque após o isolamento é considerado como imune logo não há necessidade de o integrar a aplicação de rastreio.

Notificação de possibilidade de contaminação

Na sequência da comunicação de contaminação por um utilizador, é da responsabilidade do servidor notificar todos os clientes com os quais este tenha estado em contacto. Para tal, é inicializada uma thread no servidor que irá chamar o método *verificaInfetados()* da classe *GestInfo.* Este método irá verificar todas as localizações pelas quais o User infetado andou e o intervalo de tempo que este se manteve em cada localização. Depois para cada intervalo de tempo irá analisar onde é que cada um dos outros utilizadores esteve, caso um utilizador (ou mais) tenha estado na mesma localização ao mesmo tempo que o infetado, ou seja, na mesma zona que o infetado esteve, durante esse intervalo de tempo, é então necessário enviar uma notificação para o Cliente respetivo (ou Clientes).
Para ser possível enviar as mensagens para os diferentes clientes, foi necessário guardar todas as *TaggedConection* associadas ao nome do utilizador correspondente. Para que um cliente possa receber esta notificação a qualquer momento, sem um número limitado de vezes, foi necessário implementar a execução de uma thread (no cliente) em paralelo com a principal que esteja sempre a receber mensagens do socket com a tag correspondente (6).

Descarregar mapa - Utilizador especial

Esta é a funcionalidade extra dos utilizadores especiais, sendo que ao descarregar o mapa, têm acesso ao número de infetados e de não infetados que visitaram um dado local.
Para realizar esta tarefa, bastou aceder ás estruturas Map<String,User> presentes na classe GestInfo. Sendo que cada user possui um histórico das localizações por onde passou, foi apenas necessário para cada utilizador verificar as localizações por onde passou e incrementar um contador referente a cada uma dessas localizações. É importante também referir que os utilizadores especiais são exatamente iguais aos normais sendo que a sua única diferença é que têm uma flag equivalente a 1 que os sinaliza, permitindo desta forma o acesso a esta funcionalidade extra.

Conclusão

Ao longo deste relatório, foi explicado todo o processo de criação e desenvolvimento do projeto da UC de Sistemas Distribuídos.

Para a realização deste trabalho foi necessário usar programação com *sockets* uma vez que o propósito de todas a aplicação era que os clientes se pudessem enviar pedidos ao servidor. Por outro lado, utilizou-se o controlo de concorrência de forma a que os diferentes *users* pudessem aceder á aplicação em simultâneo sem que isto implicasse erros ou incoerências de dados.

Com este projeto, o grupo consolidou todo o conhecimento adquirido ao longo desta UC, nomeadamente o desenvolvimento de software em ambiente *multi-thread*.

Concluindo, o grupo considera que realizou este trabalho prático com sucesso, uma vez que o programa responde a todas as tarefas requeridas. Além disso, referimos também que este projeto foi desenvolvido tendo em mente a versatilidade do mesmo, de modo a poder ser aplicado a diferentes mapas, zonas e locais.