Universidade do Minho

Mestrado Integrado em Engenharia Informática

Trabalho Prático - Grupo 41

Sistemas Distribuídos

Alarme COVID

Eduardo Jorge Santos Teixeira, A84707
Joana Castro e Sousa, A83614
João André Faria de Freitas, A83782
Tiago Taveira Gomes, A78141

25 de janeiro de 2021

Introdução

O Trabalho Prático do corrente ano letivo da cadeira de Sistemas Distribuídos destina-se à implementação de um serviço de rastreio de contactos e deteção de concentração de pessoas.

Para isto tem que existir dois tipos de intervenientes, o cliente e o servidor, efetuando a comunicação através de sockets e threads. Não se está a considerar a privacidade de dados e, por este motivo, podem ser livremente comunicados e armazenados no servidor.

Começamos por dividir o projeto em 3 partes distintas, sendo estas o Cliente, a App e o Server.

Cliente

A classe **"Cliente"** surgiu de forma a termos dois canais a trabalhar com o servidor em simultâneo através de sockets: um para tratar de comunicar com o servidor para a execução das funcionalidades que este oferece; e outro para receber notificações de localizações e alertas da plataforma. Consequentemente, foi então utilizada uma conexão para o suporte de diferentes padrões de interacão por processos multi-threaded (clientes e servidores), através das classes **"Demultiplexer"** e **"Tagged Connection"**. Na classe **"Tagged Connection"** dispomos da troca de dados através do conteúdo em cada classe "Frame". Esta última identifica uma etiqueta seguida de dados. Ainda assim, a classe **"Demultiplexer"** encontra-se responsável pelo delegamento de envio de mensagens para a **"Tagged Connection"**, visto que esta última está envolvida na mesma. Além disso, disponibiliza a operação *receive(int tag)* que obtém a entrada pedida e que trabalha com uma queue de mensagens para ler dessa própria entrada. No caso de não existirem mensagens nessa queue, ela passa a estar bloqueada, caso contrário, retira a primeira mensagem. Ainda na mesma classe temos a operação *send(int tag, byte[] data)*, que possibilita a transferência de dados pelo canal de saída. Não obstante, na classe **"Cliente"**, incluimos a classe **"Menu"** que comunica com o **"Demultiplexer"** para que sejam disponibilizados todos os métodos que facilitam a interação de um utilizador com os serviços da aplicação, de entre os quais, a atualização automática da localização dos diversos Clientes e a receção de várias notificações relativas às funcionalidades pretendidas.

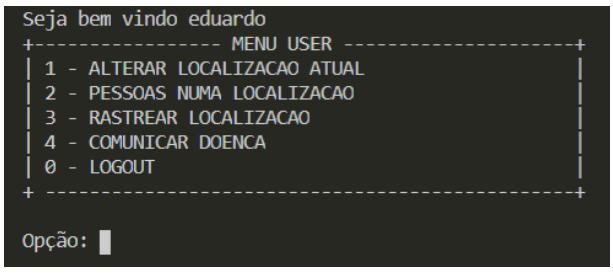

Figura 1: Menu Principal do User

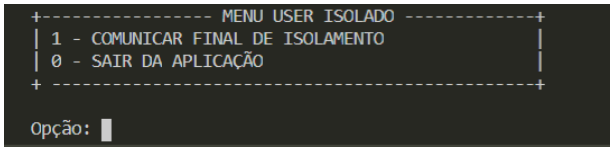

Figura 2: Menu de User Isolado

Servidor

A classe **"Servidor"** é onde se tratam novas ligações de clientes ao servidor e onde se inicializa os dados da aplicação (*APP*), da qual iremos desenvolver na seguinte secção deste documento.

Sempre que existir uma nova ligação ao servidor, este cria uma thread associada a um *Worker*, onde é estabelecida uma conexão com o *Socket* do cliente pela qual irá comunicar com o respetivo *Demultiplexer*. Este *Worker*, que ficou denominado de **"ResponseWorker"**, será o responsável pela receção da funcionalidade que o cliente pretende executar e pela resposta do server ao cliente. Também, irá ter acesso a todos os dados que estão em memória na nossa aplicação e efetuar a gestão destes mesmos dados.

Assim, em suma, um *Servidor* irá criar um *Worker* por cada cliente, que ficará encarregado que gerir todos os pedidos desse mesmo cliente, tal como notificar o cliente sempre que necessário.

App

Por fim, nesta última parte, podemos encontrar os métodos base da nossa app. Estes métodos são os responsáveis por todas as operações que modificam e acedem à estrutura da nossa plataforma. Estas são feitas em exclusão mutúa, nos objetos e variáveis necessárias, através do uso de locks e, respetivos unlocks, pois neste sentido garantimos um controlo de concorrência. Na nossa classe **App** decidimos implementar as seguintes variáveis de instância:

```
private Mapa mapa; // localizações
private Map<String, String> users; // usernames + passwords
private Map<String, ResponseWorker> usersAtivos; // users ativos
private Map<String, String> doentes; // mapa para guardar doentes
private Map<String, List<Map.Entry<Integer, Integer>>> historico_users;

// lock da app
private ReentrantLock lock;
```

Figura 3: Variáveis de instância da classe App

A variável **mapa** irá conter a estrutura de dados para definir todas as localizações existentes, na forma de uma lista de localizações (*List<Localizacao>*). Cada localização irá conter uma lista de nomes de utilizadores ativos nesse local e uma lista de todos os utilizadores que lá passaram. De forma a não ter uma lista predefinida de localizações na nossa aplicação, esta lista cresce conforme novos pedidos dos clientes, por exemplo, quando são alteradas as coordenadas de um cliente ou um cliente tentar rastrear uma localização. Caso a localização com essas coordenadas não exista, é criada uma nova.

A variável **usersAtivos** identifica um cliente com o seu respetivo apontador do *ResponseWorker*. Isto permite-nos notificar outros clientes quando um *ResponseWorker* identifica que o seu cliente ficou doente. Assim, um *worker* irá selecionar os *workers* de todos os clientes que tiverem estado na mesma localização deve cliente infetado e cada um irá notificar o seu próprio cliente.

Também, a variável **historico_users** associa a todos os clientes, todos os pares de coordenadas onde esse mesmo cliente já esteve.

O método abaixo representado **atualizaCoordUser**, através de uma THREAD_2 com tag igual a 2, permite o serviço de atualização automática das coordenadas da localização dos diversos Users/Clientes. Antes de alterarmos e verificarmos algo na nossa app adquirimos o lock que garante a execução da região crítica em exclusão mútua. De seguida, é feita a verificação do estado do User na App (Online ou não). No caso de se encontrar online é feita a remoção do User da Localização antiga e a adição deste na nova, esta passa a estar atualizada. Se porventura, este User não tem um histórico de localizações necessário para posteriores funcionalidades, é lhe criado um novo histórico e começa por ser adicionado a Localização atual. Por fim, o lock é libertado, ou seja, acontece o unlock e o valor de sucesso retornado.

```java
/**
 * Função que atualiza Localizacao do User
 *
 * @param x    - Coordenada x
 * @param y    - Coordenada y
 * @param user - User a atualizar
 * @return boolean dependendo se a função foi bem realizada com sucesso
 */

public boolean atualizaCoordUser(int x, int y, String user) {
    boolean val = false;
    try {
        lock.lock();
        if (this.usersAtivos.containsKey(user)) {
            this.mapa.removerUser(user);
            this.mapa.adicionaUser(user, x, y);
            if (this.historico_users.containsKey(user)) {
                if (!historicoTemLocal(user, x, y)) {
                    this.historico_users.get(user).add(new AbstractMap.SimpleEntry<>(x, y));
                }
            } else {
                List<Map.Entry<Integer, Integer>> array = new ArrayList<>();
                array.add(new AbstractMap.SimpleEntry<>(x, y));
                this.historico_users.put(user, array);
            }
            val = true;
        }
    } finally {
        lock.unlock();
    }
    return val;
}
```

Figura 4: Método que atualiza as coordenadas automaticamente

Cliente Especial

Nesta funcionalidade adicional, foi necessário a criação de uma nova classe denominada de **Cliente_Especial**, devendo-se ao facto, de este tipo de User conseguir um Mapa de todas as localizações contidas na nossa aplicação atual. Esta classe usufrui apenas de um tipo de thread (MAIN_THREAD) com tag 1, que ao dar *request* da operação especial, o Server fornece *reply* da funcionalidade que a um Cliente normal não é possivel. Semelhante a um user normal, este possui as classe "Demultiplexer"e "TaggedConnection"com os mesmos objetivos e métodos. De maneira a que este seja diferenciado de um Cliente normal, esta classe faz conexão com uma socket, do servidor, diferente.

Como se trata de um cliente com apenas um pedido, não foi realizado nenhum menu para este cliente e, então, mal este receba a resposta, irá fechar a sua ligação com o servidor.

```java
public class Cliente_Especial {
    private static final int MAIN_THREAD = 1;

    Run | Debug
    public static void main(String[] args) throws Exception {
        // cria socket relativo a este User
        Socket s = new Socket("localhost", 20000);
        // cria e estabelece uma conexão com o server, através de um demultiplexer
        Demultiplexer dem = new Demultiplexer(new TaggedConnection(s));
        dem.start();

        try {
            // send request
            dem.send(MAIN_THREAD, "BACKUP>".getBytes());
            // get reply
            byte[] data = dem.receive(MAIN_THREAD);
            // print da resposta
            System.out.println(new String(data));

            // fechar dem
            dem.close();
        } catch (Exception ignored) {
            //
        }
    }
}
```

Figura 5: Classe do Cliente Especial

O Diagrama a seguir exposto retrata de uma forma mais ilustrativa e exemplificativa de todo o processo do nosso projecto, sendo assim possível fazer uma imagem mais global para um melhor entendimento de todo o processo.

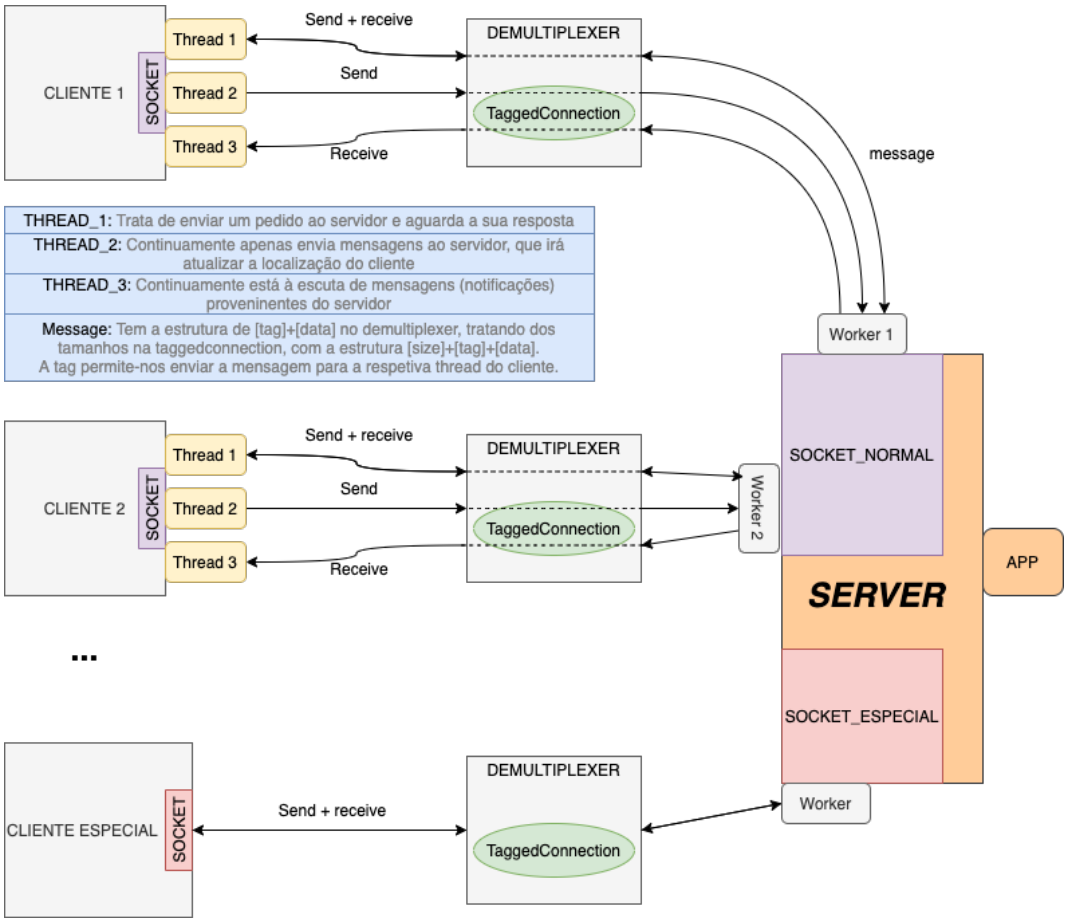

Figura 6: Diagrama exemplificando o processo

Conclusão

Em suma, após a finalização deste projeto, conseguimos pôr em prática os conceitos lecionados nas aulas de Sistemas Distribuídos, estabelecendo um protocolo universal para a troca de mensagens, conjugado com a possibilidade do servidor responder a diversos clientes simultaneamente. Para isso foi necessário recorrer a *multithreading* e, consequentemente, a controlo de concorrência e esperas passivas, o que nos permitiu realizar este trabalho com sucesso.

Finalizando, terminamos este trabalho prático com todas as funcionalidades pretendidas e conhecimentos anteriormente adquiridos, agora ainda mais consolidados.