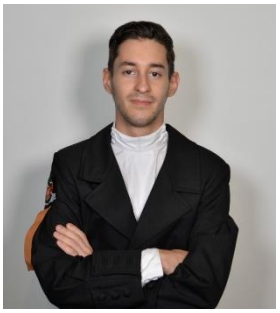
Licenciatura em Engenharia e Gestão de Sistemas de Informação

FUNDAMENTOS DE SISTEMAS DISTRIBUÍDOS 2022/2023

3ª FASE

PL3_G5



Gonçalo Fernandes Pereira

A96550

Joana Marília Pinto Sousa Pereira

A95539

Mariana Magalhães Araújo

A97245

Implementação

Este trabalho consiste na criação de um sistema de partilha de mensagens de texto com uma arquitetura cliente-servidor que permite a troca de mensagens privadas entre clientes e a troca de mensagens seguras, isto é, que garantem autenticidade, entre os clientes.
Para a realização deste optamos por criar um servidor central, o qual todos os clientes se irão conectar. Para cada cliente o servidor irá criar um agente de utilizador que será responsável por toda a comunicação que será através do sockets TCP.

Servidor.java

Esta classe é responsável por criar o servidor ao qual todos os clientes se vão conectar e armazenar toda a informação necessária, como a lista de presenças, de clientes e mensagens. Além disso, tem como variáveis de instância a porta na qual o servidor será iniciado e o tempo que cada utilizador poderá estar inativo.

Esta classe é constituída pelo método *verInativos()* que permite obter uma lista de todos os utilizadores que se encontram inativo.

O método *sessionTimeout()* é responsável por enviar aos clientes que se encontram inativos uma mensagem SESSION_TIMEOUT e encerrar os buffers e a conexão com esses clientes.

Por fim, o método *timerTimeout()* corresponde a um Timer que, de 5 em 5 segundos, irá verificar os clientes inativos e desconecta-los.

Quando o servidor é executado, inicia as respetivas listas e o *timerTimeout()* e fica a aguardar a conexão de clientes. Quando algum se conectar, inicia uma thread ClienteHandler que fará a gestão desse cliente.

Cliente.java

Esta classe corresponde ao programa do cliente e é constituída por duas variáveis de instância, host e porta, que permitem que o cliente se conecte ao servidor.

Quando o cliente é executado, o socket é iniciado juntamente com o BufferReader que irá permitir receber as mensagens do servidor e um PrintWriter que permite enviar mensagens para o servidor. Assim que o cliente se conecta é enviada uma mensagem do tipo SESSIONS_UPDATE_REQUEST ao servidor contendo o seu username permitindo, assim, estabelecer a conexão.

Uma vez que o cliente recebe e envia mensagens, simultaneamente, é criada uma thread, que é executada enquanto o socket estiver conectado, permitindo receber as mensagens enviadas pelo servidor. Estas mensagens são divididas em cabeçalho, que contem o protocolo da mensagem, e em corpo da mensagem que corresponde ao conteúdo da mensagem.

Se a mensagem for do tipo SESSION_UPDATE, a mensagem será processada e mostrada ao cliente. Se a mensagem começar com "/*cliente*/", a mensagem é dividida e obtemos as informações relativas a cada cliente sendo armazenadas nos respetivos HashMaps permitindo associar o respetivo IP a cada cliente e a respetiva chave publica.

Caso seja do tipo SESSION_TIMEOUT, os buffers são fechados e a conexão com o servidor é terminada.

De forma a permitir enviar mensagens, enquanto a conexão estiver estabelecida, o Cliente fica em loop à espera de que algo seja escrito e envie para o servidor uma mensagem AGENT_POST. Em cada cliente é executado um Timer de 60 em 60 segundos que envia ao servidor um SESSION_UPDATE_REQUEST que permite manter a sessão do cliente ativa.

Para implementar o suporte RMI, quando o cliente é iniciado é criado o registo na porta 1099 e a interface remota do cliente. De seguida, no registo criado é guardada a referência remota com o nome SERVICE_NAME, isto é "/PrivateMessaging".

Foi necessário também realizar alterações quando o cliente escreve alguma mensagem. Com esta nova funcionalidade, quando o Cliente escrever "/msg <username> <mensagem>" irá ser possível enviar uma mensagem privada a um determinado cliente.
Para tal, obtemos o IP do cliente destino, acedemos ao seu registo e obtemos o registo cujo nome é SERVICE_NAME, obtendo assim a referência remota ao cliente destino. Por fim, o método remoto é executado, enviando assim a mensagem ao cliente.

No caso de o método retornar "false", significa que o cliente não quer receber mensagens privadas. Em caso contrário, irá retornar o username do destinatário.

Por fim, para enviar mensagens seguras, inicialmente geramos um par de chaves, isto é uma chave publica e uma chave privada. A chave publica é enviada para o servidor quando o cliente se conecta de forma a este a poder difundir para os restantes clientes.

Para enviar uma mensagem deste género o cliente deve escrever "/msgSegura <username> <mensagem>" e, para que seja enviada garantindo autenticidade, inicialmente geramos um sumário da mensagem com o algoritmo SHA-256. Esse sumario é assinado com a chave privada do remetente e posteriormente convertido para Base 64 para ser enviado. Por fim recorremos ao método sendSecureMessage da interface PrivateMessagingSecure.

ClienteHandller.java

Esta classe corresponde à thread que faz a gestão dos diferentes clientes. É constituída por diversos métodos que permitem fazer esta gestão.

O método *sessionUpdate()* permite construir a mensagem SESSION_UPDATE contendo os clientes ativos e respetivo IP, chave publica e o valor logico que indica se quer receber mensagens ou não, e as ultimas 10 mensagens enviadas. De seguida, enviá-la para o cliente recebido como argumento.

O método *lerProtocolo()* é responsável por receber uma determinada mensagem do cliente e dividi-la no protocolo e no corpo da mensagem. Se a mensagem recebida for AGENT_POST, atualiza a atividade do cliente remetente, adiciona a mensagem à lista de mensagens e difunde para todos os clientes ativos.

Cada thread irá receber a informação do servidor de forma a mantê-la atualizada e vai criar dois buffers para permitir a comunicação entre ambos. Quando iniciada, irá ficar a aguardar a comunicação do cliente e processar essa mensagem através do *lerProtocolo().*

IPInfo

Esta classe tem como objetivo guardar as informações dos clientes tais como o IP, a data da sua última vez ativo e o username.
É constituída pelo método *timeOutPassed()* que recebe o valor do SESSION_TIMEOUT e retorna true ou false consoante se ele está, ou não, ativo.

ListaPresencas.java

Esta classe fará a gestão da atividade de todos os clientes que se conectarem ao servidor.
Toda a informação será guardada numa Hashtable que tem como chave o endereço IP de um determinado cliente e como valor um objeto IPInfo.
O método *getIPList()* retorna um Vector que contém a lista de todos os IPs que estão ativos.
O método *getPresences()* permite atualizar a atividade do IP recebido como argumento, mesmo sendo a primeira conexão daquele Cliente, e retorna a lista fornecida pelo *getIPList().*

PrivateMessagingSecureInterface

Permite a criação de uma interface remota para um cliente possibilitando assim a troca de mensagens diretas entre clientes.
Esta classe define o método sendMessage que recebe como argumentos o username do utilizador que enviou a mensagem privada, o conteúdo da mensagem e retorna o username do utilizador que recebeu a mensagem.
Também define o método sendMessageSecure que recebe como argumentos o username do utilizador que enviou a mensagem privada, o conteúdo da mensagem enviada e a assinatura, isto é, o sumário da mensagem assinado. Este método retorna o username do utilizador que recebeu a mensagem.

PrivateMessagingSecure

Corresponde à implementação da interface PrivateMessagingSecureInterface.
Tem como variáveis de instância o username e o valor lógico receberPM permitindo guardar na interface o username e a possibilidade, ou não, do cliente receber mensagens privadas.
Além disso contém um HashMap que associa a cada cliente a sua chave publica, o que nos permite verificar a assinatura do sumário enviado.
O método sendMessage é definido e irá escrever no terminal do cliente a respetiva mensagem enviada.
É também definido o método sendMessageSecure que obtém a chave publica do cliente que enviou a mensagem e com recurso a esta obtém o sumário da mensagem produzido originalmente.
De seguida geramos outro sumario da mensagem original e comparamos os 2 de forma a verificar se são iguais. Em caso afirmativo, significa que a mensagem não recebeu alterações, ou seja, está autenticada, sendo mostrada ao cliente destinatário.

Questões

Fase 1

Q1: Especificações abertas

Para conseguir desenvolver um agente de utilizador que consiga interagir com o nosso servidor é necessário que este seja capaz de receber uma LinkedList que armazena todos os clientes, uma ListaPresencas que faz a gestão de toda a atividade dos clientes e uma ArrayList que contem as mensagens AGENT_POST difundidas pelo servidor.
Como toda a comunicação é feita através de sockets TCP é necessário que o agente de utilizador, quando recebe algo do servidor, consiga verificar o cabeçalho dessa mensagem.

Q2: Falha de componentes

Na realização do sistema é feito um tratamento de todas as exceções que podem ser lançadas quer pelo servidor, quer pelo cliente através da estrutura Try/Catch.
Contudo, após alguns testes ao sistema, este não faz o correto tratamento da falha no caso de um Cliente se desconectar inesperadamente, o que origina um erro no servidor fazendo-o falhar. Além disso, quando um cliente recebe o SESSION_TIMEOUT, ocorre um erro no buffer que não está a ser corretamente tratado. Porém, este erro não afeta o correto funcionamento nem do servidor, nem do cliente.
Apesar a realização de todos os testes ao programa, poderão existir mais erros que nós não detetamos.

Q3: Limitações do sistema

Uma limitação do desenho do sistema que consideramos afetar o desempenho e a escalabilidade deste sistema é a implementação através de sockets TCP, uma vez que no caso de existir um grande número de clientes conectados, a difusão de mensagens entre todos os clientes será mais dispendiosa em termos de recursos da máquina pois existe um canal de comunicação exclusivo com cada um.

Q4: Difusão de mensagens pelo servidor

Serem os clientes a tomar a iniciativa de contactar o servidor para saber se há nova informação seria mais simples de implementar uma vez que o pedido realizado por um cliente a um servidor é mais simples que a resposta que o servidor vai dar a esse pedido, isto é, enquanto o cliente apenas envia um pedido para se conectar ao servidor, este responde com a lista de clientes online e com a lista das mensagens mais recentes. No entanto nesse caso a informação não estaria atualizada para todos os clientes ao mesmo tempo, o que seria resolvido se o servidor notificasse regularmente todos os clientes sempre que ocorresse uma alteração.

Q5: Implementação em sockets UDP

Este projeto poderia ser implementado com base em sockets UDP, contudo o protocolo TCP é mais recomendado uma vez que toda a comunicação entre o servidor e os clientes é feita através de mensagens e este protocolo garante que todas as mensagens são entregues e na respetiva ordem. Isto é, através do uso do TCP conseguimos garantir que a ordem das mensagens enviadas no SESSION_UPDATE está correta. Por outro lado, se fosse esperado um grande número de clientes conectados ao servidor, o protocolo UDP poderia torna-se mais adequado.

Fase 2

Q1: Especificações abertas

Para que seja possível a troca direta de mensagens entre diferentes clientes através do java RMI é necessário que inicialmente o cliente se conecte ao servidor por uma ligação via sockets, porém quando esta conexão é estabelecida é criada uma refencia remota a este cliente que é armazenada no seu registo na porta 1099 que lhe permite comunicar com os restantes clientes.

Q2: Difusão de mensagens pelo servidor

Para se tornar proveito do RMI para que fosse possível o servidor tomar a iniciativa de notificar todos os clientes sempre que ocorrer uma alteração no estado do sistema seria necessário que o servidor guardasse no seu registo os endereços IP's de todos os clientes conectados. Quando existisse uma entrada, saída ou mensagem nova, o servidor tomaria a iniciativa de enviar uma mensagem privada a todos os clientes com o novo estado da sessão.

Q3: Alternativa com sockets

Para que fosse possível implementar a troca de mensagens diretas sem recorrer ao RMI, mas através do uso de sockets, seria necessário o servidor guardar os Buffers de entrada e saída de cada cliente. Quando um cliente enviasse uma mensagem privada com um determinado destinatário, o servidor seria responsável por reencaminhar essa mensagem para o buffer de saída de determinado cliente, permitindo assim a troca de mensagens diretas sem recurso ao RMI.

Fase 3

Q1: Confidencialidade

A criptografia de chave pública poderia ser implementada para assegurar a confidencialidade das mensagens por RMI, uma vez que, cada entidade que é suposto ter acesso aos recursos possuiu 2 chaves geradas em conjunto, uma que não pode ser revelada a ninguém (chave privada) e uma que é para divulgação pública (chave pública). Não é possível determinar a chave privada com base na chave pública e as mensagens cifradas com uma chave podem ser decifradas com a outra, isto é, uma mensagem cifrada com a chave pública de um utilizador, só poderá ser decifrada com a chave privada deste mesmo utilizador.

Q2: Confiança

Com o uso da criptografia da chave pública, não é garantido que a chave pública pertença realmente ao cliente em questão. Os certificados digitais poderão ser uma solução alternativa para mitigar estas vulnerabilidades, uma vez que são documentos digitais que atestam a associação entre uma chave pública e um indivíduo ou entidade. Este fornece identificação a uma entidade, é resistente à falsificação e é possível verificar que foi emitido por uma entidade oficial e de confiança. Os certificados digitais contêm os seguintes elementos: chave pública, nome da entidade que é certificada (proprietário da chave), nome da autoridade certificadora, nº de série, data de emissão e validade, outra informação de certificação (ID, foto, etc.) e assinatura digital da entidade emissora (assinatura digital do certificado).