Sistemas Distribuídos

Aplicação Chat

Mestrado Integrado em Engenharia Informática e Computação

Inês Teixeira
Maria João Mira Paulo
Nuno Miguel Mendes Ramos
Pedro Duarte da Costa

8 de Julho de 2017

Conteúdo

1 Introdução

Este trabalho consiste na criação de uma aplicação *Chat*, cliente-servidor, capaz de conectar utilizadores, criar *chats* de grupo com um ou mais participantes, permitindo-lhes a troca, tanto de mensagens como de ficheiros.

2 Arquitetura

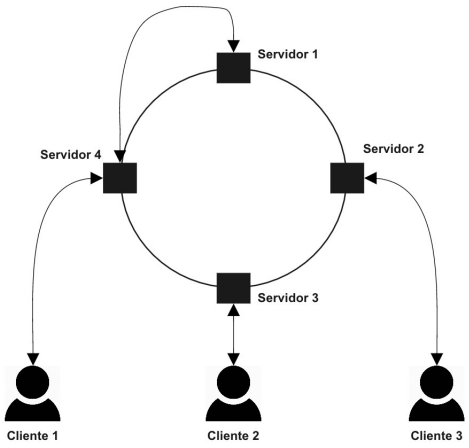

Figura 1: Arquitetura Base do programa

2.1 Servidor

Os servidores têm três grandes responsabilidades:

- **Armazenamento dos dados**: Cada servidor é responsável por uma gama de utilizadores e deve armazenar as informações de *login* (*email* e *password*), os *Chats* e as Chaves públicas e privadas de cada cliente.
- **Distribuição dos dados**: Além de guardar a informação dos seus utilizadores, o servidor é ainda responsável por guardar em *backup* a informação dos utilizadores do servidor que o precede na *Distributed Hash Table*. É ainda responsável por garantir um grau de replicação de dois. Em caso de falha de um servidor, o seu sucessor deve guardar os dados em *backup* e enviar para o seu sucessor, de forma a manter o grau de replicação desejado.
- **Controlo de mensagens**: Por forma a facilitar a comunicação cliente-servidor, cada servidor é responsável por tratar das mensagens dos seus utilizadores. Caso contrário deve redirecionar as restantes mensagens para o respetivo servidor. Desta forma, cada cliente pode comunicar com qualquer servidor sendo que a sua mensagem chegará, de forma eficiente, ao servidor correto.

2.1.1 *Distributed hash table*

Foi implementada uma *Distributed hash table* usando o protocolo/algortimo *Chord* com o objetivo de obter uma distribuição equilibrada dos dados pela

rede de servidores, proporcionar uma rápida pesquisa de servidores e troca de mensagens entre os mesmos. O algoritmo de *Chord*, distribui

$$2^m$$

servidores numa rede circular virtual ordenada armazenando pares chave e valor, respetivamente o identificador do servidor e o servidor em questão. Por forma a obter uma rápida transmissão de mensagens e procura de servidores, cada um destes possui uma *Finger Table*. Esta armazena **m** apontadores, sendo que o primeiro elemento da *Finger Table* aponta para o sucessor imediato. As restantes posições da *Finger Table* apontam para o servidor responsável por todos os clientes cujo o identificador se encontra entre

$$n + 2^{i-1} \tag{1}$$

e

$$n + 2^i \tag{2}$$

. N é a posição do servidor a efetuar o calculo e i a posição da *Finger Table*. Desta forma, sempre que um servidor precisar de encontrar o responsável por um determinado pedido apenas têm que verificar na *Finger Table* o servidor que sucede a chave do pedido em questão. De seguida deve redirecionar o pedido para esse servidor até que este seja o sucessor imediato do que recebeu o pedido.

Desta forma é possível evitar uma pesquisa linear. Em vez de pedido pela rede de sucessor em sucessor até que seja possível encontrar o desejado é feita uma pesquisa que contacta O(log N) servidores.

2.2 Cliente

O *Cliente* tem a possibilidade de criar uma conta na aplicação, *Sign Up* ou entrar numa conta criada anteriormente, *Sign In*.

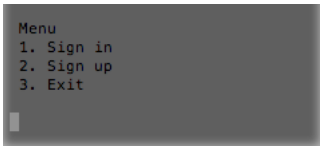

Figura 2: Menu Inicial

Assim que o cliente cria uma nova conta ou acede à sua conta, já existente, a aplicação redireciona o cliente para o Menu Principal. Aqui é permitido ao utilizador a criação de um novo *Chat*, a abertura de um dos seus *Chats*, o envio de ficheiros e o *download* de ficheiros recebidos.

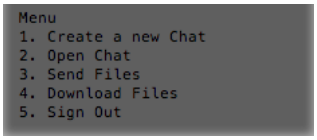

Figura 3: Menu Principal

Em ***Create a New Chat***, é permitido ao Cliente a criação de um *Chat* com um ou mais participantes, convidados através do seu email.

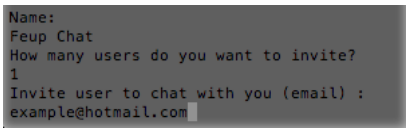

Figura 4: Criação de um novo *Chat*

De seguida o cliente pode começar a conversar com os participantes.

Figura 5: Listagem de *Chat* do utilizador

Além disso, em ***Send Files*** o cliente pode enviar um anexo para um determinado *Chat*. Caso o cliente tenha recebido um anexo num dos seus *Chats* pode fazer download do ficheiro em ***Download Files***, no Menu Principal.

O Cliente tem acesso, em cada um dos seus *Chat*, às suas novas mensagens e às mensagens lidas. Além disso, todas as mensagens contêm a sua data de criação e o email do utilizador que a enviou, como pode ser visto na imagem a seguir.

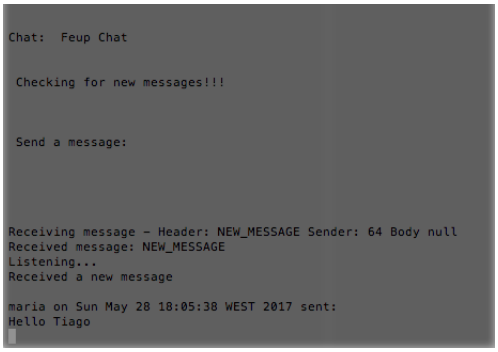

Figura 6: Troca de mensagem num *Chat*

2.3 Estrutura de Classes

Para uma melhor estrutura do programa foram construídas as seguintes classes:

- **Node**: Representa um *node* na *Distributed hash table*, contendo o identificador do mesmo, *IP* e respetiva Porto.

- **Server**: Derivada da classe *Node*. Representa um Servidor na rede. Contém informação relativa aos seus utilizadores, informação relativa à *Distributed hash table*, aos utilizador que se encontram *online* e um *backup* dos dados do servidor que o precede na rede.

- **User**: Responsável pela informação relativa ao utilizador como o *e-mail* e *password*. Além disso contém duas estrutura de dados que guardam tanto *chats* aos quais o utilizador pertence, tal como os pedidos pendentes, ou seja, pedidos ainda não vistos pelo utilizador.

- **Client**, derivada da classe *User*. Armazena apenas informação relativa aos seus *Chats*.

- **Chat**: Responsável por armazenar o nome do *Chat* e respetivo identificador, identificador do criador do *Chat*, um *Array List* de Mensagens e um *Array List* de Mensagens de *Chat* Pendentes, ou seja, ainda não vistos pelo utilizador.

- **ChatMessage**: Representa uma Mensagem enviada através do *Chat* e é responsável por guardar o identificador do *Chat* ao qual pertence, a data de criação da mesma, o conteúdo, o *e-mail* do remetente e o tipo de mensagem, texto ou qualquer outro tipo de ficheiro.

- **Message**: Responsável por guardar informação quanto ao tipo de Mensagem, identificador do remetente, *String* que remete para a condição de responsabilidade pela mensagem e o conteúdo da mesma, podendo este ser um conjunto de *Strings* ou um objeto *Serializable*.

- **Connection**: Garante a comunicação cliente-servidor e servidor-servidor. Esta classe contém uma *SSLSocket* responsável por estabelecer a comunicação, um *ObjectInputStream* reponsável por receber mensagens e um *ObjectOutputStream* capaz de enviar mensagens.
  Concluindo, esta classe contém os métodos que permitem o envio de mensagens, a receção de mensagens e, tanto a inicialização como o fecho de conexões.

- **Server Connection**: Derivada da Classe *Connection*. Usada pelo Servidor.

- **Client Connection**: Derivada da Classe *Connection*. Usada pelo Cliente.

- **Distributed hash table**: Contém métodos capazes de conectar *nodes* e estabilizar a rede.

- **Utilities**: Contém métodos de geração de uma *hash* a partir de um certo dado, métodos de encriptação e desencriptação de dados e geração de chaves públicas e privadas.

- **Constants**: Contém constantes necessárias à implementação do programa, como nomes de mensagens e possíveis tipos de erros.

A classe *Server Connection* e a *Client Connection* derivam da classe *Connection*. Esta estrutura foi escolhida pela simplificação do processamento das mensagens recebidas, uma vez que as mensagens recebidas pelo Cliente diferem das recebidas pelo Servidor.

3 Implementação

3.1 Distribuição dos servidores

A distribuição dos servidores foi conseguida recorrendo à implementação de uma *Distributed Hash Table*, como referido na secção da Arquitetura.

3.1.1 *Distributed Hash Table*

A implementação da *Distributed Hash Table* foi pensada de maneira a apresentar uma solução distribuída e escalável dos servidores.
Esta contém até 2^{32} servidores e 2^{256} clientes. Esta implementação foi feita através da criação uma *hash* do *Ip* e da Porto do servidor e, no caso do cliente, através do seu *e-mail*.
Foi utilizado o algoritmo *SHA-256* (*Utilities.Utilities* linha 21) com o objetivo de garantir 2^{256} identificadores distintos. De maneira a identificar o servidor com um valor possível de armazenar num inteiro truncamos a *hash* calculada para 32bits (*Utilities.Utilities*, linha 47). Os identificadores dos clientes são armazenados em variáveis do tipo *BigInteger*, uma classe *Java*, que permite armazenar um número de qualquer dimensão. A função mais importante da *DHT* é a *nodeLookUp(int key)* (*Protocols.DistributedHashTable*, linha 96). Esta função recebe o identificador do *node* ou cliente que pretendemos localizar na rede.

3.1.2 *Node LookUp*

Esta função (Protocol.DistributedHashTable - linha 96) é utilizada sempre que se pretende "saltar"entre servidores na *distributed hash table*. Esta função procura qual é o "melhor salto"dependendo da situação.

Existem três casos possíveis:

- A *finger table* não dá a volta - este caso é o mais simples visto que, basta procurar na *finger table* qual a posição onde o valor é menor que a chave que se pretende procurar e a posição seguinte é maior;
- A *finger table* dá a volta mas a chave não - neste caso procura-se na finger table o primeiro valor maior que a chave ou o primeiro valor após dar a volta. Para testarmos se tal acontece, bastar fazer uma subtração entre dois nós consecutivos da *finger table*, e caso dê negativo significa que deu a volta.
- A *finger table* dá a volta e a chave também - neste caso vamos descartando nós, enquanto não dermos a volta à *finger table*, usando o teste explicado no ponto anterior. Após dar-mos a volta procuramos pelo primeiro valor maior que a chave.

Para sabermos que a *finger table* dá a volta fazemos a subtração do *ID* do último elemento da *finger table* com o *ID* do nó atual. Caso dê negativo, significa que tal aconteceu. E para finalizar, para sabermos que a chave dá a volta, fazemos a subtração do último elemento da *finger table*.

3.1.3 *Join Network*

Inicialmente quando o primeiro servidor da rede se liga, recebe nos seus argumentos o seu *IP* e Porto, e inicializa a sua *Finger Table*, ocupando ele

próprio todas as posições da mesma, e as estruturas de dadas necessárias para guardar os utilizadores.

Os restantes servidores no seu construtor recebem como parâmetros o seu *IP* e o seu porto, e ainda o *IP* e porto de um servidor já existente na rede. Estes ligam-se à rede chamando a função *joinNetwork(this, knownNode)*. Esta recebe o próprio servidor e o servidor já existente e envia uma mensagem, *NEWNODE*, para o servidor existente a avisar da sua entrada na rede.

O servidor que recebe a mensagem *NEWNODE* verifica na *Finger Table* através da função *nodeLookUp(key)* qual o servidor que sucede na *Finger Table* o novo servidor, não garantindo que seja o imediato sucessor na *DHT*. Esta mensagem é enviada pela *DHT* até encontrar o sucessor imediato do novo servidor (*Server.Server*, linha 245). Quando tal se verifica, o servidor envia a sua *Finger Table* para o novo *node* (*Server.Server* , linha 236), de forma a que este obtenha uma perceção do estado atual da rede, notifica o servidor que o precede da existência de um novo *node* que será o sucessor imediato do novo servidor (*Server.Server*, linha 236) e envia o backup dos dados do seu antigo predecessor para o novo *node* (*Server.Server*, linha 237).

Finalizado este processo o servidor que acabou de se ligar à rede já possui uma *Finger Table* informada e o *backup* do servidor que o precede. Desta forma a rede mantém-se informada e é sempre possível encontrar o servidor responsável por um certo cliente, pois está sempre atualizado sobre o seu sucessor.

3.2 Protocolo de *Backup*

Este protocolo tem como objetivo prevenir a perda de informação. Este é acionado sempre que o utilizador envia informação para o servidor (credenciais, *chats* e mensagens). Aquando da chegada de nova informação, o servidor guarda pela primeira vez ou atualiza a sua estrutura de dados que contém os utilizadores:

```
1
2 private ConcurrentHashMap<BigInteger, User> users;
```

Após guardar/atualizar a sua estrutura, o servidor utiliza a mensagem BACKUP_USER para avisar o seu sucessor para atualizar ou guardar o utilizador na sua estrutura de dados que contém os *backups*:

```
1
2 private ConcurrentHashMap<BigInteger, User> backups;
```

Foi escolhido o envio da informação para o seu sucessor, uma vez que, caso este servidor, por alguma razão, se desligue, será o seu sucessor que ficará responsável pelos seus utilizadores. A informação é guardada com um *replication degree* de dois.

3.3 Protocolo de *Node Failure*

Um cliente deteta que o servidor responsável por ele falhou, assim que este tenta enviar-lhe uma mensagem. Nesse momento é iniciado o protocolo de *node failure* do cliente.

Este protocolo consiste na tentativa do cliente conectar-se com outro servidor. O cliente tem a possibilidade de receber nos seus parâmetros de inicio do programa, um Ip e Porto extra, para tentar recuperar a conexão.

```
1
2 private int recoverServerPort;
3 private String recoverServerIp;
```

Caso não receba estes parâmetros, o cliente no início do programa passa o valor do I p e porto original para o Ip e porto de recuperação, visto que o servidor final do cliente pode não ser o inicial, logo poderá ter sido outro que foi abaixo.

Caso conseguia conectar-se com o servidor de recuperação, o cliente envia a mensagem SERVER_DOWN com o id do servidor que foi abaixo. Após o envio desta mensagem, o cliente espera um segundo para os servidores atualizarem as suas *finger tables*. Quando volta a executar, o cliente tenta fazer SIGN_IN e acabará, provavelmente, por estabelecer comunicação com o sucessor do servidor que foi abaixo, visto que nesse momento é ele o responsável pelo cliente e é ele que contém os *backups* do servidor que foi abaixo. Nesse momento, o novo responsável pelo cliente, passa os valores da estrutura dos **backups** para a estrutura dos seus utilizadores.

Um servidor deteta que outro falhou, quando ele se tenta conectar com ele e não consegue. Após se aperceber disso, o servidor chama a função serverDown(Node downNode) (ficheiro Server.java, linha 1000) que manda a mensagem SERVER_DOWN, para os restantes servidores atualizarem as suas *finger tables*.

3.4 Troca de Mensagens

A aplicação é conseguida através da contínua troca de mensagens entre cliente-servidor e servidor-servidor. Para isso, e como referido anteriormente, foi criada a Classe **Message**, sendo esta uma classe **Serializable**.

De forma a tornar o processo troca de mensagens mais simples e eficaz foi implementado um processo de Serialização. Este processo consiste na tradução de estruturas de dados num formato (*array* de *bytes*) que possa ser armazenado e reconstruido posteriormente.

Para isso, as classes *Message, Chat, ChatMessage, User, Server e Node* implementam a classe **Serializable**.

```
1  public void sendMessage(Message message) throws IOException {
2
3      if (message == null)
4              throw new IOException();
5       outputStream.writeObject(message);
6
7  }
8
9  public Message receiveMessage() throws IOException,
       ClassNotFoundException {
10
11     return (Message) inputStream.readObject();
12
13 }
```

Toda a comunicação é realizada com recurso a *SSL Sockets*. Tanto o cliente com o servidor são autenticados através do uso de chaves públicas e privadas e com recurso a certificados armazenados no disco, da mesma forma que foi implementado no Lab5.

3.4.1 Redirecionamento dos Pedidos

Como mencionado anteriormente, o cliente não se liga necessariamente ao servidor responsável. Todas as mensagens têm um campo onde diz se o servidor

que a recebe, é ou não responsável pelo cliente. Caso este campo diga que não é responsável, tais como nas mensagens de *Sign_ In* e *Sign_ Up*, o servidor irá fazer *lookup* na *finger table* da chave do cliente, e caso encontre o servidor responsável pelo cliente, a mensagem é reencaminhada a dizer que o próximo servidor é responsável, senão apenas reencaminha para o servidor que dê o maior salto na *finger table*, que irá repetir o processo.

Quando o servidor responsável é encontrado, este executa o pedido do cliente e envia a mensagem de resposta, que vai seguir o caminho oposto da primeira mensagem, chegando assim ao cliente. O cliente vai reparar que o ip e porto de quem enviou a mensagem é diferente do servidor para onde enviou inicialmente, sabendo assim o ip e porto do servidor responsável por ele, logo ele fecha a conexão atual e abre uma nova com este servidor.

3.5 Comunicação Cliente Servidor

O cliente recebe nos parâmetros do programa o Ip e Porto de um servidor ativo.

Contudo, a comunicação cliente-servidor apenas é iniciada quando um utilizador cria conta ou inicia sessão. Nesse momento é criado um *hash* com o email do utilizador. O *hash* é utilizado para encontrar o servidor responsável pelo cliente.

Este processo começa quando o servidor recebe uma mensagem de SIGNIN ou SIGNUP, nesse momento o servidor irá fazer *lookup* do *hash* na sua *finger table* pelo valor. Se encontrar o nó responsável pelo cliente, este reenvia-lhe a mensagem, senão reenvia para o maior servidor onde faz o maior salto.

3.6 Mensagens Cliente Servidor

Quanto à comunicação Cliente Servidor foram criadas diferentes mensagens, tais como:

- **Client_Success**: Mensagem de sucesso.
- **Client_Error**: Mensagem de erro.
- **New_Chat_Invitation**: Mensagem recebida quando o cliente recebe um convite de participação para um novo *Chat*. Esta mensagem contém o objecto *Chat*.
- **New_Message**: Mensagem recebida quando o cliente recebe uma nova *Chat Message* num determinado *Chat*. Esta mensagem contém o objeto *Chat Message*.
- **Downloading_File**: Mensagem recebida quando o servidor envia um ficheiro pedido pelo cliente.
- **Server_Update_Connection**: Mensagem recebida pelo cliente quando um novo servidor é adicionado à rede e torna-se responsável por este. Esta mensagem contém o *IP* e Porto do novo servidor.
- **Server_Success**: Mensagem de sucesso do servidor.
- **Server_Error**: Mensagem de erro do servidor.

3.7 Mensagens Servidor Servidor

Quanto à comunicação Servidor Servidor foram criadas diferentes mensagens, tais como:

- **Sign_In**: Mensagem recebida quando o utilizador pretende iniciar sessão, contém o email e a *hash* da palavra-passe.
- **Sign_Up**: Mensagem recebida quando o utilizador pretende criar sessão, contém o email e a *hash* da palavra-passe.
- **Sign_Out**: Mensagem recebida quando o utilizador termina sessão.
- **Create_Chat**: Mensagem recebida quando o utilizador pretende criar um novo chat, contém o objeto *chat*.
- **Create_Chat_By_Invitation**: Mensagem recebida quando um *chat* é criado e o convidado não está no servidor que recebeu o pedido *Create_ Chat*.
- **Get_Chat**: Mensagem recebida quando um utilizador pretende obter um *chat*.
- **Get_All_Chats**: Mensagem recebida quando um utilizador pretende obter todos os seus *chats*.
- **Get_All_Pending_Chats**: Mensagem recebida quando um utilizador pretende obter todos os seus *chats* pendentes.
- **Invite_User**: Mensagem recebida quando um cliente convida outro cliente para um *chat*.
- **New_Message_To_Participant**: Mensagem recebida quando um utilizador envia uma nova mensagem e o cliente não está no servidor que recebeu o pedido inicial.
- **New_Message**: Mensagem recebida quando um utilizado envia uma nova mensagem para um *chat*, contém o *ID* do *chat*.
- **New_Node**: Mensagem recebida quando um novo servidor entra na rede.
- **Predecessor**: Mensagem que avisa que o servidor é o predecessor de um determinado servidor.
- **Successor_Ft**: Mensagem que contém a *finger table* do predecessor.
- **Backup_User**: Mensagem usada para fazer *backup* de um utilizador, contém um objeto *User*.
- **Add_User**: Mensagem usada para adicionar um utilizador, contém um objeto *User*.
- **User_Updated_Connection**: Mensagem usada para atualizar a conexão do servidor.
- **Server_Success**: Mensagem de sucesso do servidor.
- **Server_Down**: Mensagem de erro do servidor.

- **File_Transaction** : Mensagem responsável pela transferência de um ficheiro.
- **Store_File_On_Participant**: Mensagem responsável pelo armazenamento de ficheiros do lado do cliente.
- **Store_File_Message**: Mensagem responsável pelo armazenamento de ficheiros.
- **Download_File**: Mensagem responsável pelo download de um ficheiro.

3.8 Interface

De forma a tornar a aplicação *user-friendly* foi implementada uma Interface em *Java Swing*. No desenvolvimento da interface foi usado o *Model-View-Control Pattern* de forma a que a lógica fosse independente da interface. Ou seja, foi criado uma classe *Controller.java* que cria um objeto da classe *Client.java* e um objeto da classe *InterfaceView.java*.

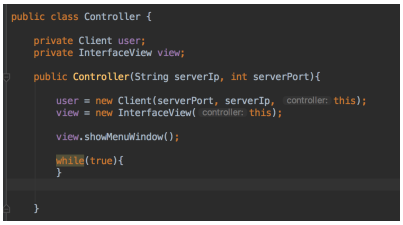

Figura 7: Interface: Construtor *Controller.java*

Figura 8: Interface: Construtor *InterfaceView.java*

Figura 9: Interface: Construtor *Clientr .java*

O Menu Inicial, o *Sign Up*, o *Sign In*, o Menu de um utilizador ligado e a criação de um *chat* com vários utilizadores foram criados com sucesso.

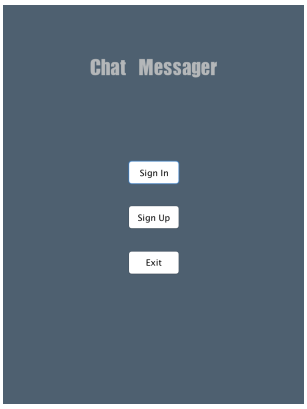

Figura 10: Interface: Menu inicial

O utilizador com o rato escolhe a opção de Sign In, Sign Out ou Exit. Ou seja, entra na sua conta, cria conta ou sai do programa.

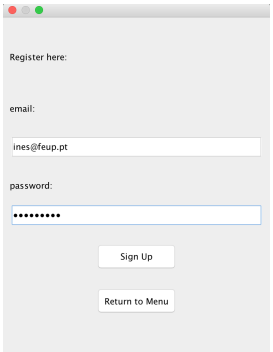

Figura 11: Interface: Sign In

O utilizador insere as suas credenciais e entra na sua conta. Durante este processo há uma mudança de estado na *Task* do cliente de WAITING_SIGN_IN para SIGNIN. Caso seja feita com sucesso, senão esta mudança de estado não ocorre e é lançado uma janela de notificação e pedido novamente as credencias.

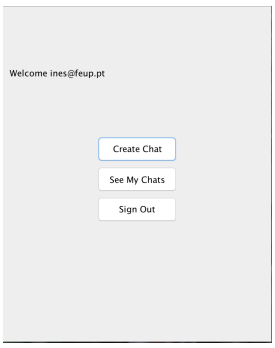

Figura 12: Interface: Menu utilizador logado

Se o utilizador criar com sucesso a sua conta então este é levado para uma janela onde pode criar um *chat*, ver os seus *chats* ou terminar sessão.

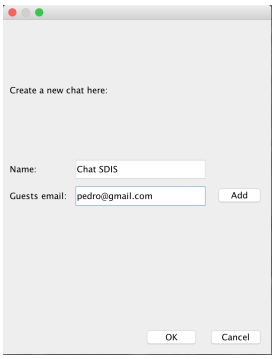

Figura 13: Interface: Criar *chat* com múltiplos utilizadores

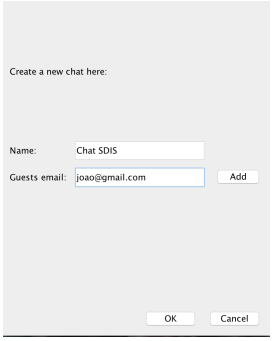

Figura 14: Interface: Criar *chat* com múltiplos utilizadores

Como é possível ver na imagem, na criação de um novo *chat* é possível adicionar vários utilizadores tal como no programa principal clicando no botão *Add* conforme adiciona um novo convidado.

Figura 15: Interface: Sala de *chat*

Esta última janela, foi apenas idealizada, não tendo sido possível aparecer

todas as mensagens em cadeia na janela de mensagens.

4 Aspetos Relevantes

4.1 Transmissão de ficheiros

Um aspeto relevante do projeto, relativamente ao Cliente, foi a transmissão de ficheiros entre clientes e servidores. Esta transmissão foi feita dividindo o ficheiro em *chunks* de 8192 bytes, tamanho máximo limite na comunicação SSL.

4.2 Função de *hash* utilizada na *DHT*

Utilizamos o algoritmos SHA-256 para gerar os vários identificadores utilizados na DHT por ser um algoritmo bastante seguro até à atualidade, que permite a criação de um elevado número de identificadores diferentes e é um algoritmo de encriptação constante. Este ultimo ponto é bastante relevante para a obtenção de uma distribuição de dados uniforme na DHT, permitindo balançar os dados de forma relativamente equitativa por todos os servidores.

4.3 Encriptação

Foi implementado um sistema de geração de chaves públicas e privadas para cliente. Estas são geradas, no primeiro *sign up*, com base na password do cliente e guardadas no servidor respetivo. A chave pública é guardada sem qualquer proteção extra. A chave privada é encriptada usando a *password* do cliente. Desta forma o cliente pode fazer *login* noutro computador obter a sua chave pública e privada, sendo que esta ultima apenas pode ser obtida pelo utilizador pois para ser desencriptada precisa da password do mesmo.

Após criação de um *chat* ou *recepção* de um novo, o cliente em questão envia para os participantes do *chat* a sua chave pública. Por falta de tempo, não foi possível finalizar a encriptação do conteúdo das mensagens que seguiria a seguinte lógica: sempre que algum participante escreve uma mensagem, são criadas cópias da mesma por cada participante e encriptado o seu conteúdo de acordo com a chave pública do respetivo participante.

5 Conclusões

Concluindo, teria sido interessante a finalização de toda a interface gráfica e a implementação completa da encriptação de mensagens. Estas duas implementações não foram concluídas devido à falta de tempo.

Além disso, foi pensado que, como possíveis melhorias, estariam a eliminação de *Chats*, a eliminação de participantes de *Chats* e eliminação de mensagens.