UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS DE QUIXADÁ SEMESTRE 2021.2

Relatório - Sistema Agenda

Disciplina: Sistemas Distribuídos

Equipe: Daniel Vitor - 472213, Petrucio Neves - 469854, Samuel Henrique - 473360

Professor: Marcos Dantas

QUIXADÁ - CE

2022

1. Visão geral do serviço remoto

Este sistema se apresenta como um protótipo para serviços de uma agenda de contatos. O objetivo do desenvolvimento foi implementar algumas funcionalidades onde são utilizadas informações sobre contatos. Os métodos possibilitam adicionar, editar, procurar, remover, listar e limpar agenda.

O sistema foi implementado em Java, tanto no servidor quanto no cliente, e utilizando a arquitetura cliente-servidor e usando sockets UDP, de modo que os métodos possam ser utilizados remotamente de forma transparente pelo usuário.

Além disso, foi usado o ProtocolBuffer para definir uma interface de comunicação heterogênea.

2. Descrição dos Métodos Remotos

Primeiramente, foi definido quais atributos formam os objetos que são usados como parâmetros dos métodos remotos.

- **Telefone:**
  - telefone: String
  - tipo: Enum(Mobile, Personal, Home, Work)
- **Endereço:**
  - endereço: String
  - tipo: Enum(Home, Work)
- **Email:**
  - email: String
  - tipo: Enum(Personal, Work)
- **Contato:**
  - nome: String
  - telefones: List<Telefone>
  - endereços: List<Endereço>

  - emails: List<Email>

- **Agenda:**
  - contatos: List<Contato>

Após isso, foram definidos os métodos remotos do sistema.

- **Adicionar Contato:**
  - **In:** Contato
  - **Out:** Boolean (true caso o nome já esteja sendo usado, false caso contrário)
  - **Exceções:** Nome não pode ser vazio
- **Listar Contatos:**
  - **In:** Void
  - **Out:** List<Contato>
  - **Exceções:** Vazio
- **Buscar Contatos:**
  - **In:** Contato
  - **Out:** List<Contato>
  - **Exceções:** Vazio
- **Editar Contato:**
  - **In:** Agenda
  - **Out:** Boolean (true caso tenha sido editado, false caso contrário)
  - **Exceções:** Nome não pode ser vazio
- **Remover Contato:**
  - **In:** Contato
  - **Out:** Boolean (true caso tenha sido removido, false caso contrário)

  - **Exceções:** Vazio

- **Limpar Agenda:**
  - **In:** Void
  - **Out:** Void
  - **Exceções:** Vazio

3. Descrição dos Dados transmitidos

Com a definição dos métodos, foi possível realizar a modelagem do sistema através da utilização do diagrama de classe UML. Dessa forma, as classes atuam em três diferentes vertentes: cliente, servidor e modelos. A Figura 1 e a Figura 2 abordam as partes do cliente, onde são descritas as classes “User”, “Proxy” e “UDPClient”. A Figura 3 ilustra as classes referentes ao servidor, as quais podem ser listadas por “UDPServer”, “Despachante”, “AgendaEsqueleto” e “SistemaAgenda”. Por fim, a Figura 4 mostra as classes relacionadas aos modelos.

Figura 1 - Diagrama de Classe (Cliente)

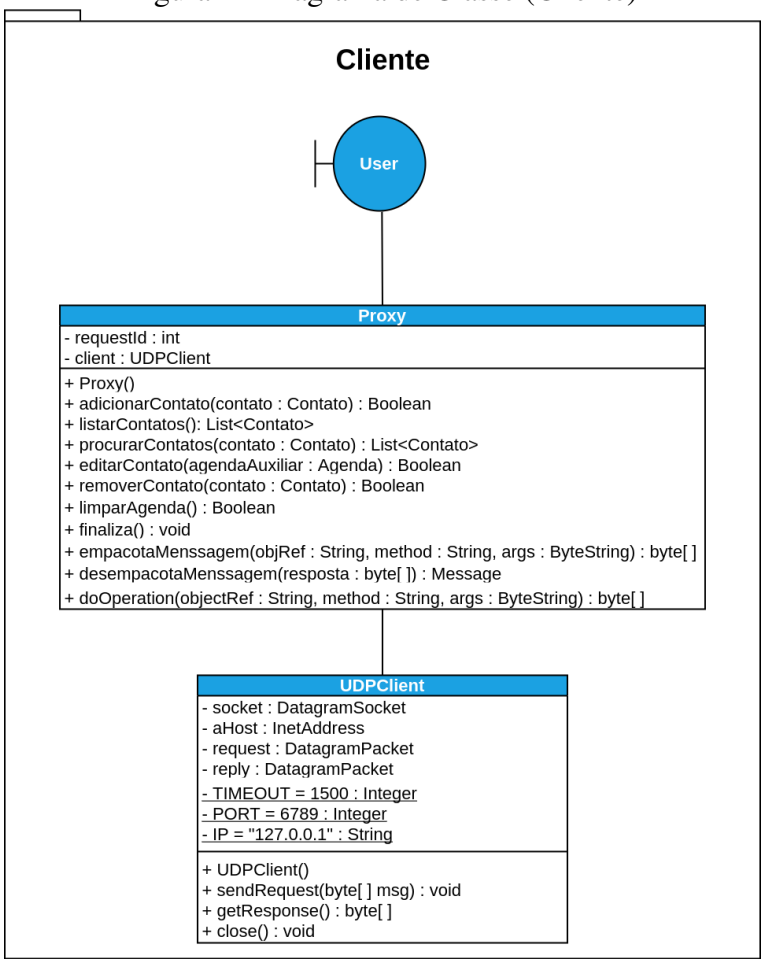


Figura 2 – Diagrama de Classe (Outros)

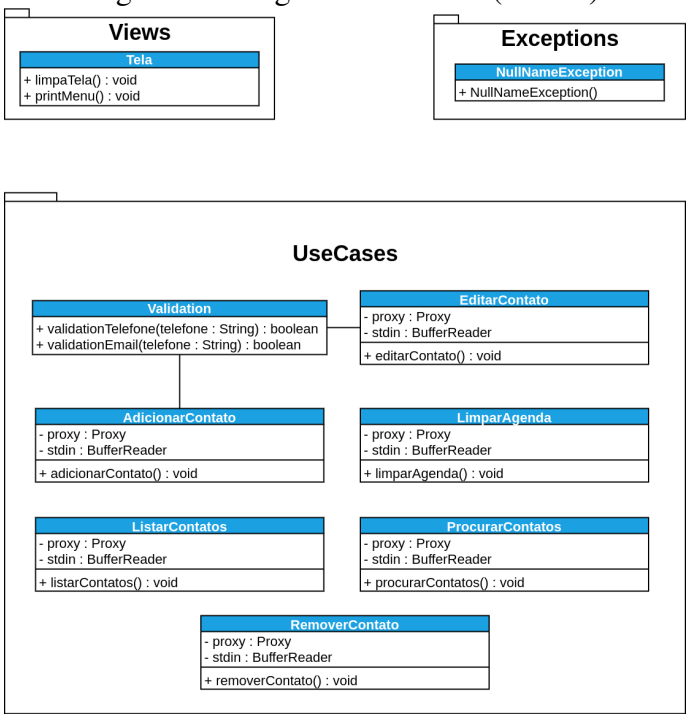


Figura 3 – Diagrama de Classe (Servidor)

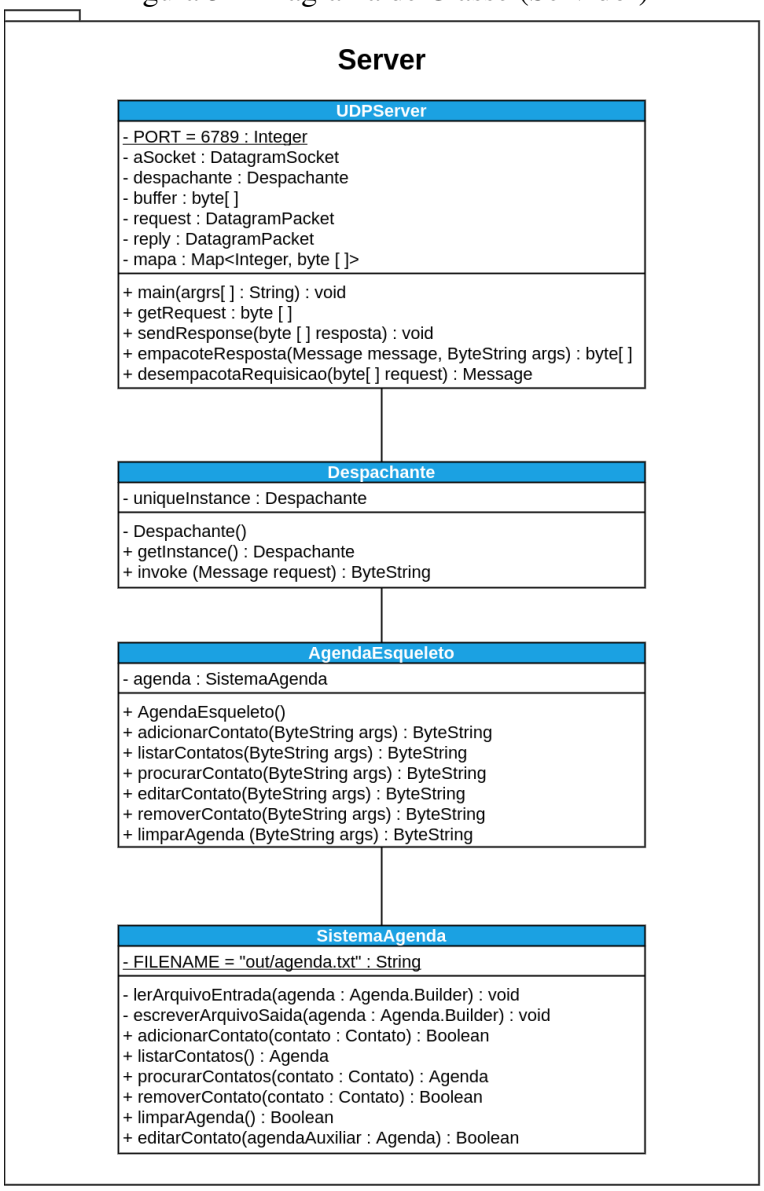
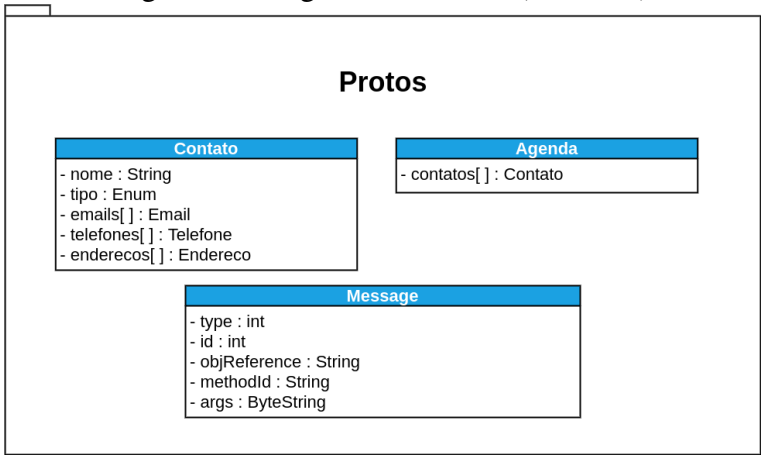


Figura 4 – Diagrama de Classe (Modelos)



3.1 Dados Transmitidos

Foi usada a representação de dados do Protocol Buffer para transmitir dados via Socket UDP. Dessa forma, é possível visualizar abaixo as descrições sobre as classes, assim como suas implementações.

- **Message**

A classe **Message**, ilustrada na figura, é responsável por encapsular a requisição que será transmitida via Socket UDP. Essa classe contém todas as informações necessárias para realizar uma requisição, possuindo o id da requisição, o objeto e método escolhido, o tipo e os argumentos. A implementação pode ser visualizada na **Figura 5**.

Figura 5 - Implementação da Classe Message

```
syntax = "proto3";

package com.trabalhoFinal.protos;

option java_package = "com.trabalhoFinal.protos";
option java_outer_classname = "MessageProto";

message Message {
  int32 type = 1;
  int32 id = 2;
  string objReference = 3;
  string methodId = 4;
  bytes args = 5;
}
```

- **Agenda**

A classe **Agenda**, ilustrada na figura, é responsável por representar os dados referentes a Agenda e sua implementação pode ser visualizada na figura **Figura 6**.

Figura 6 - Implementação da Classe Agenda

```
syntax = "proto3";

package com.trabalhoFinal.protos;

option java_package = "com.trabalhoFinal.protos";
option java_outer_classname = "AgendaProto";

message Contato {
  string nome = 1;

  enum Type {
    MOBILE = 0;
    PERSONAL = 1;
    HOME = 2;
    WORK = 3;
  }

  message Email {
    optional string email = 1;
    optional Type type = 2;
  }

  message Telefone {
    optional string telefone = 1;
    optional Type type = 2;
  }

  message Endereco {
    optional string endereco = 1;
    optional Type type = 2;
  }

  repeated Email emails = 2;
  repeated Telefone telefones = 3;
  repeated Endereco enderecos = 4;
}

message Agenda {
  repeated Contato contatos = 1;
}
```

4. Descrição das classes implementadas

Do lado do cliente temos as classes **User**, **Proxy**, **UDPClient**, e as classes nos pacotes **useCases**, **views** e **exceptions**.

A classe **User** é responsável por fazer a interface do usuário com o sistema, nessa classe temos uma instância do **Proxy** o qual vamos utilizar para fazer todas as chamadas das funções.

A classe **Proxy** é responsável por abstrair para o User as chamadas ao servidor principal. Além de empacotar e desempacotar as mensagens a serem enviadas/recebidas via **UDP**, e serializar as requisições e desserializar as respostas.

A classe **UDPClient** é responsável por efetuar a comunicação com o servidor trocando os dados que foram empacotados anteriormente.

As classes do pacote **useCases** são responsáveis por receber os dados nescessários do usuário e realizar a chamada ao **Proxy** para fazer as requisições ao servidor. As regras de negócio estão concentradas nas classes desse pacote.

A classe do pacote **views** é chamada pela classe **User** e é responsável por apresentar o menu ao usuário via terminal.

A classe **NullNameException.java**, do pacote **exceptions**, é responsável por retornar uma exceção caso o usuário digite um nome vazio do contato.

Do lado do servidor temos as classes **UDPServer**, **Despachante**, **AgendaEsqueleto** e **SistemaAgenda**.

A classe **UDPServer** é responsável por desempacotar e desserializar as mensagens, atender as requisições, empacotar e serializar as respostas e depois enviar de volta a resposta daquela requisição.

A classe **Despachante** é responsável por decifrar qual o método está sendo chamado através daquela requisição e utilizar a classe **AgendaEsqueleto** para executar tais métodos, através do método reflexivo. Além disso, ele é responsável por retornar a saída que os métodos invocados retornaram.

A classe **AgendaEsqueleto** é responsável por fazer o intermédio entre o **Despachante** e o **SistemaAgenda**. O **AgendaEsqueleto** executa o método do sistema e após isso retorna em um ByteString para que seja enviado posteriormente como resposta da requisição solicitada.

A classe **SistemaAgenda** é responsável por implementar todos os métodos do sistema da Agenda para serem executadas em um servidor. Toda a parte da persistência de dados está concentrada nessa classe.

5. Descrição do modelo de falhas

Ao desenvolver sistemas distribuídos, precisamos ter em mente que a internet não é um meio perfeito, onde todas as requisições são atendidas em um tempo curto, e sim um meio onde as requisições se perdem, atrasam, se corrompem e assim por diante. Com isso, precisamos criar um modelo de falhas para poder lidar com as inconsistências que a rede mundial de

computadores possa sofrer.

Um erro comum é a perda de mensagens de requisição e/ou resposta. Para tratar isso, foi definido um timeout de 1,5s no processo cliente para que, se uma mensagem não for respondida nesse tempo, ela será retransmitida.

Uma nova requisição só será enviada quando a resposta da requisição atual for respondida. Por causa disso, o sistema é passível de receber requisições duplicadas. Mediante isso, tratamos a repetição de mensagens no lado do servidor. Armazenamos a requisição e o seu ID em uma estrutura chamada TreeMap, na qual o ID é armazenado como uma chave e os bytes da requisição como valor. Ao receber uma requisição com um ID já visto anteriormente, a resposta é enviada sem acesso ao sistema.

6. Como executar

As instruções de execução podem ser encontradas no arquivo *out/README.md*.

7. Conclusão

O trabalho mostrou-se bastante enriquecedor pois foi possível colocar em prática vários conhecimentos adquiridos durante a cadeira de Sistemas Distribuidos, como arquitetura Servidor-Cliente, modelo de falhas, protocolo no formato requisição-resposta, métodos reflexivos, serialização/desserialização, uso da IDL ProtocolBuffers, etc.

Com o método reflexivo, usado na classe **Despachante**, foi possível perceber a facilidade de adicionar e remover métodos, e como isso ocorre de forma transparente para o servidor.