UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS DE QUIXADÁ

Relatório

Serviço Remoto de Votação

Disciplina: Sistemas Distribuídos – 2022.2

Equipe:

Rafael Galdino da Silva – 495101

Eliton Lima Rosendo Filho – 493508

Igor Gabriel – 496458

Quixadá - CE.

1. **Visão geral do serviço do remoto**

   O serviço remoto que nossa equipe desenvolveu se trata de um sistema de votação. Onde um usuário, após se autenticar, pode realizar um único voto. Além disso, é possível também listar os votos para ver quem está vencendo. O tópico da votação é “Qual linguagem de programação você prefere?", porém o sistema pode ser utilizado para diversos outros tipos de votação ou enquete.

2. **Descrição dos Métodos Remotos**

   Método 1 – Login: Esse método recebe um nome de usuário e uma senha, esses valores precisam estar cadastrados previamente no “banco de dados” (No exemplo, estamos utilizando apenas um arquivo .csv para salvar dados) e então após autenticar corretamente o usuário pode realizar outras chamadas de métodos remotos

   Método 2 – Votar: Nesse método, o usuário autenticado escolhe sua opção de voto (representada por um número), o sistema então processa esse voto e salva no “banco de dados” (arquivo .csv), caso o usuário já tenha realizado uma votação, o sistema o impede de votar novamente.

   Método 3 – Listar votos: Nesse método, o sistema lista todos os votos registrados de uma forma legível para o usuário final com o intuito de saber quem está ganhando e perdendo a votação.

3. **Descrição dos Dados transmitidos**

   A execução do programa de inicia pela classe *Client*, que é responsável exclusivamente pela interação com o usuário, aqui o usuário irá selecionar através de um menu interativo qual ação deseja realizar no sistema, sendo 1 para votar, 2 para listar votos e 3 para sair.

   A depender da opção selecionada, o sistema pode requisitar que o usuário se autentique, requisitando um nome de usuário e uma senha.

A depender do método remoto que será solicitado, uma classe proxy equivalente é chamada – *AuthProxy* para requisições de autenticação e *VoteProxy* para requisições relacionadas a votar ou listar votos. Ambas herdam de uma classe *Proxy,* que tem método implementado para empacotar os dados recebidos em um *JSONObject* (da biblioteca *org.json*) e enviar esses dados empacotados através da classe *UDPClient,* que envia esses dados ao servidor.

Do lado do servidor, a classe UDPServer é responsável por receber as mensagens e chama o método *invoke()* classe *Dispatcher.* Essa classe é responsável por desempacotar a mensagem, que irá informar qual o serviço e método remoto que deve ser chamado.

Em seguida, um respectivo esqueleto é chamado – *AuthSkeleton* para requisições de autenticação e *VoteSkeleton* para requisições de votação ou listagem de votos. Cada um desses esqueletos chama uma classe de serviço relacionada (*AuthService* e *VoteService*, respectivamente). Essas são as classes que fazem a alteração no armazenamento permanente.

As classes do lado do servidor são responsáveis por empacotar uma resposta, também no formato *JSONObject*, que será retornada para o *UDPClient*.

A seguir, uma visualização do formato de uma **requisição:**

```
{
 "request_id" : "b900431c-4a5e-4f8b-9ed9-24835ab4f55a",
 "service" : "Vote",
 "operation": "vote",
 "args" : {
          "option" : "java"
 }
}
```

request_id: UUID tipo 4 pseudo-aleatório (gerado pela classe *java.util.UUID*).

service: Sinaliza o serviço remoto que será chamado pelo Dispatcher.

operation: Sinaliza o método remoto que deve ser chamado.

args: Um segundo *JSONObject* que envia os argumento necessários para o serviço.

A seguir, uma visualização do formato de uma **resposta:**

```
{
 "request_id" : "b900431c-4a5e-4f8b-9ed9-24835ab4f55a",
 "status": "403",
 "message": "Usuário já votou!"
}
```

request_id: O mesmo ID enviado pela requisição, serve para controlar o fluxo de envio de requisição e resposta.

status: Status de erro/sucesso, inspirado pelos Códigos de status de respostas HTTP.

message: Mensagem da resposta

OBS: No exemplo acima, o usuário já estaria previamente autenticado, do contrário não seria possível a realização do voto.

4. **Descrição das classes implementadas nos lados Cliente e Servidor**

No lado do Cliente temos a classe *Client* para interação com o usuário, as classes *Proxy, AuthProxy, VoteProxy* para empacotamento de mensagem (em JSON), e a classe *UDPClient* que é a classe responsável por enviar e receber pacotes do servidor.

Do lado do Servidor, temos a classe *UDPServer*, que recebe as mensagens e envia respostas. Por padrão o servidor está aberto na porta 3000. Em seguida temos a classe Dispatcher, responsável por desempacotar a requisição e realizar a chamada do método remoto através dos esqueletos (*AuthSkeleton* e *VoteSkeleton*) que respectivamente receberão os argumentos da requisição necessários para chamar o serviço respectivo (AuthService ou *VoteService*), o

método do serviço também é definido na mensagem e tratado pelo despachante (O serviço de votação possui duas possibilidades de método remoto, votar ou listar votos).

Ademais, do lado do servidor temos a classe *CSVUtil*, que é responsável por gerenciar os dados armazenados permanentemente dentro do arquivo .csv utilizando a biblioteca *com.opencsv* (**Não estamos utilizando um banco de dados, essa é uma solução temporária**).

Outros métodos que são importantes na troca de mensagens entre Cliente e Servidor são:

doOperation() – Método disponível na classe *Proxy* (que é herdado por *AuthProxy* e *VoteProxy*) esse método empacota as mensagens e realiza o envio.

send() – Método da classe *UDPClient* responsável pelo envio de uma mensagem empacotada para o servidor.

receive() – Método da classe UDPClient responsável por receber uma mensagem empacotada de resposta do servidor.

invoke() – Método da classe Dispatcher que é responsável por desempacotar a mensagem do lado do servidor, permitindo a chama do serviço e método remoto equivalente.

Além disso, é importante mencionar os métodos chamados para a realização dos serviços:

login() – Método existente na classe *AuthProxy* que recebe um nome de usuário e uma senha. Esse também é o nome do método presente na classe de serviço AuthService, que checa o armazenamento (arquivo .csv) para validar o usuário através dos dados recebidos

vote() – Método da classe *VoteProxy*, recebe uma opção. Esse também é o nome do método presente na classe de serviço *VoteService*, checa se o usuário já votou, se não tiver votado realiza o voto (inserindo o voto do usuário no armazenamento per)

list() – Método da classe *VoteProxy*, não recebe argumento algum. Esse também é o mesmo nome do método presente na classe de serviço *VoteService.* Esse método tem como objetivo obter os dados de votação contidos no arquivo .csv de armazenamento permanente, e lista eles para o usuário.

5. **Descrever o modelo de falhas**

As falhas que podem ocorrer no envio de requisições ao serviço distribuído podem gerar uma exceção do tipo ***RequestFailedException***, uma exceção especial, criada para essa aplicação, que vai ser chamada dependendo do código do status recebido na resposta.

200 – Ok: Operação realizada com sucesso, não gera exceção

403 – Proibido: Operação não pode ser realizada, gera exceção

401 – Não autorizado: Erro de autenticação / Usuário não autorizado, gera exceção

Além disso, está implementado uma configuração de timeout de 1 segundo, onde caso não haja uma resposta do servidor, é gerado uma exceção ***SocketTimeoutException***, que indica para o proxy correspondente que é necessário fazer um re-envio da mensagem.

Ademais, outras exceções podem ser geradas em caso de falha ao acessar o servidor, são elas:

- I**OException:** Foi tratado em cada método um aviso deste tipo de *"Exception"* utilizando a palavra throws, definindo que o método poderia lançar aquele tipo de *"Exception"* e assim conseguir tratá-lo.

- **SocketException:** O sistema trata a *"Exception"* para qualquer problema referente ao socket criado, tanto para o cliente quanto para o servidor.

- **ClassNotFoundException:** É tratado na classe despachante, no caso da classe não encontrar a outra classe que foi passada via requisição.

- **InstantiationException:** Caso haja alguma falha ao instanciar um objeto de uma determinada classe, está exception pode ser lançada no método newInstance da classe Despachante.

- **IllegalAccessException:** É lançada a classe na classe Despachante no momento de criação da instância ao objeto.

- **NoSuchMethodException:** É tratado na classe Despachante e também pode ser lançada na mesma caso não seja encontrado o método passado via requisição.

- **SecurityException:** Caso haja alguma violação de segurança, é tratado na classe Despachante.

- **IllegalArgumentException:** É utilizada para validar valores de parâmetros, que não estejam nas condições do programa, assim ,você pode querer que um parâmetro do seu método, que é de tipo inteiro, nunca seja maior do que 50 para evitar falhas no programa.

- **InvocationTargetException:** É lançada quando um método ou um construtor de objeto gera uma exception, como os demais é tratado na classe Despachante.

- **CsvException:** No nosso exemplo, o armazenamento permanente é um arquivo .csv, essa exceção (da biblioteca *com.opencsv*) indica algum erro na leitura ou escrita do arquivo csv.

- **FileNotFoundException:** Caso o arquivo .csv não seja encontrado, essa exceção é lançada na classe *CSVUtil* (que gerencia o controle do armazenamento permanente).