Relatório - Trabalho 1 - Sistema Distribuídos
Augusto César Araújo de Oliveira - 508991
Francisco Gustavo Martins de Sousa - 511779

Questão 1:

PessoasOutputStream.java

Relatório sobre o código da classe `PessoasOutputStream`:

O código é uma implementação de uma classe em Java que lida com a saída de dados de objetos do tipo `Pessoa` em diferentes formatos. A classe herda de `OutputStream` e fornece métodos para escrever informações de pessoas em três formatos diferentes: via terminal, em um arquivo e via conexão TCP em um servidor remoto.

Aqui estão os principais aspectos do código:

- A classe possui um construtor vazio e um construtor que recebe um array de objetos `Pessoa` e um `OutputStream` como parâmetros.

- O método `writeSystem` é responsável por escrever as informações das pessoas no terminal. Ele usa um `PrintStream` temporário para escrever as informações.

- O método `writeFile` é usado para escrever as informações das pessoas em um arquivo. Ele usa um `FileOutputStream` e um `BufferedOutputStream` para escrever os dados em bytes.

- O método `writeTCP` é responsável por enviar os dados das pessoas via conexão TCP para um servidor remoto. Ele utiliza um `DataOutputStream` para enviar os dados em formato binário.

- No método `main`, são criadas duas instâncias da classe `PessoasOutputStream` para lidar com a saída de dados em diferentes formatos. A primeira instância escreve no terminal, a segunda escreve em um arquivo e a terceira envia os dados via conexão TCP para um servidor remoto.

```
            // fecha o buffer
            buffer.close();

            // mensagem de sucesso
            System.out.println(x:"Arquivo escrito com sucesso!");

        } catch (IOException error) {
            error.printStackTrace();
        }
    }

    public void writeTCP() {
        try {
            // Cria um DataOutputStream para enviar os dados
            DataOutputStream dataOutputStream = new DataOutputStream(this.op);

            // Envia o número de pessoas
            int numeroDePessoas = pessoas.length;
            dataOutputStream.writeInt(numeroDePessoas);

            for (Pessoa pessoa : pessoas) {
                // Envie o número de bytes utilizados para o nome da pessoa
                int tamanhoNome = pessoa.getNome().getBytes(charsetName:"UTF-8").length;
                dataOutputStream.writeInt(tamanhoNome);

                // Envie o nome da pessoa
                dataOutputStream.write(pessoa.getNome().getBytes());

                // Envie o CPF e a idade
                dataOutputStream.writeUTF(pessoa.getCpf());
                dataOutputStream.writeInt(pessoa.getIdade());
            }
        } catch (IOException error) {
            error.printStackTrace();
        }

    }
```

ServidorOutputStream.java

Relatório sobre o código da classe `ServidorOutput`:

O código é uma implementação de um servidor em Java que aceita conexões de clientes e lida com a entrada de dados recebida. O servidor espera por conexões de clientes em uma determinada porta e, ao estabelecer uma conexão, cria uma nova instância da classe `Connection` para tratar a comunicação com o cliente.

Aqui estão os principais aspectos do código:

- O método `main` inicia o servidor em uma porta específica (12345) e aguarda a conexão de um cliente.

- Quando um cliente se conecta, o servidor aceita a conexão e exibe o endereço IP do cliente. Em seguida, cria uma instância da classe `Connection` para lidar com a comunicação com o cliente.

- A classe interna `Connection` estende `Thread` e é responsável por lidar com a entrada de dados do cliente. Ela inicializa `DataInputStream` e `DataOutputStream` para ler e escrever dados do cliente.

- O método `run` da classe `Connection` implementa um servidor de eco simples. Ele lê a quantidade de pessoas enviadas pelo cliente, seguida pelo tamanho do nome, o nome, o CPF e a idade de cada pessoa. Em seguida, exibe essas informações no console do servidor.

```
    DataOutputStream out;
    Socket clientSocket;

    // construtor da classe recebe um socket como parâmetro
    public Connection(Socket aClientSocket) {
        try {
            // atribui o socket recebido ao atributo da classe
            clientSocket = aClientSocket;
            // cria um DataInputStream no socket
            in = new DataInputStream(clientSocket.getInputStream());
            // cria um DataOutputStream no socket
            out = new DataOutputStream(clientSocket.getOutputStream());
            // inicia a thread
            this.start();
        } catch (IOException e) {
            System.out.println("Connection:" + e.getMessage());
        }
    }

    public void run() {
        try { // an echo server
            int quantidadePessoa = in.readInt();
            System.out.println("Quantidade de pessoas: " + quantidadePessoa);
            for (int i = 0; i < quantidadePessoa; i += 1) {
                int tamanhoNome = in.readInt();
                System.out.println("Tamanho do nome: " + tamanhoNome);
                byte[] nomeBytes = in.readNBytes(tamanhoNome);
                String nome = new String(nomeBytes);
                System.out.println("Nome: " + nome);
                String cpf = in.readUTF();
                System.out.println("CPF: " + cpf);
                int idade = in.readInt();
                System.out.println("Idade: " + idade);

            }
            this.clientSocket.close();
    You, 6 days ago • update questoes 1 e 2
        } catch (EOFException e) {
            System.out.println("EOF:" + e.getMessage());
        } catch (IOException e) {
```

Questão 2:

PessoasInputStream.java

Relatório sobre o código da classe `PessoasInputStream`:

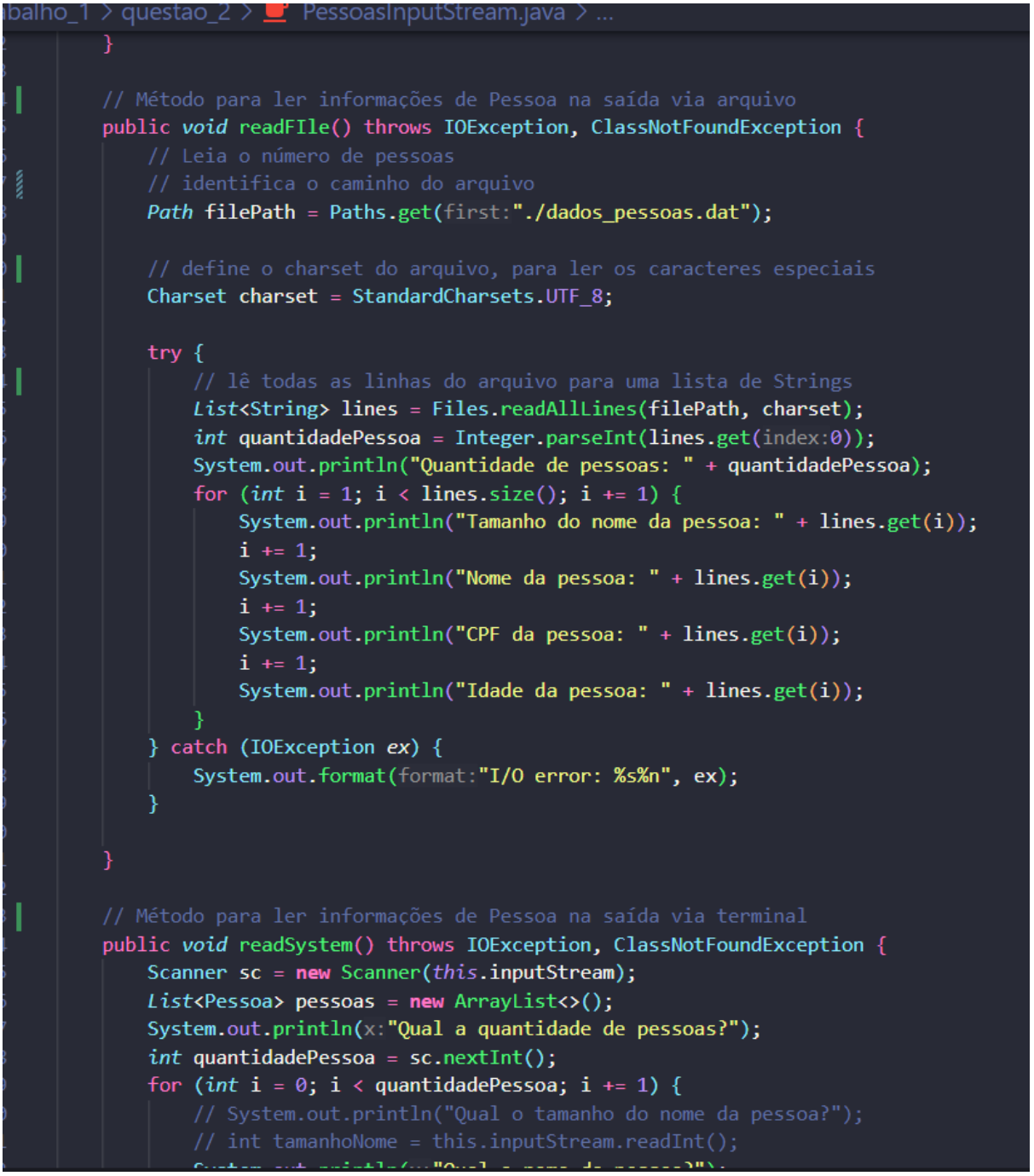
O código é uma implementação em Java de uma classe `PessoasInputStream` que lê dados de entrada de três fontes diferentes: terminal, arquivo e conexão cliente-servidor TCP. A classe estende a classe `InputStream` e fornece métodos para ler e processar dados de entrada a partir dessas fontes. Abaixo estão os detalhes sobre os principais aspectos do código:

- O construtor da classe `PessoasInputStream` recebe um `InputStream` como parâmetro e inicializa o atributo `inputStream`.

- O método `readFile` lê dados de um arquivo chamado "dados_pessoas.dat". Ele lê a quantidade de pessoas seguida pelos detalhes de cada pessoa, como tamanho do nome, nome, CPF e idade, e os imprime no console.

- O método `readSystem` lê dados de entrada a partir do terminal. Ele solicita ao usuário a quantidade de pessoas e, em seguida, solicita detalhes para cada pessoa, como nome, CPF e idade, e os armazena em objetos `Pessoa`. Em seguida, imprime os detalhes de todas as pessoas no console.

- O método `readTCP` lê dados de entrada de uma conexão cliente-servidor TCP. Ele usa um `DataInputStream` para ler a quantidade de pessoas, seguida pelos detalhes de cada pessoa, como tamanho do nome, nome, CPF e idade, e os imprime no console.



ServidorInputStream.java

Relatório sobre o código da classe ‘ServidorInputStream’

- O código começa criando um `ServerSocket` na porta 12345 e imprime "Servidor iniciado". Ele aguarda uma conexão do cliente.

- Quando uma conexão é estabelecida, o endereço IP do cliente é impresso no console, seguido pela mensagem "conexão estabelecida".

- É criada uma classe interna chamada `Connection`, que estende a classe `Thread`. Essa classe é responsável por manipular a conexão com o cliente. Ela cria um `DataInputStream` e um `DataOutputStream` para manipular os dados recebidos e enviados pela conexão.

- No método `run` da classe `Connection`, o servidor lê a quantidade de pessoas enviada pelo cliente e, em seguida, lê os detalhes de cada pessoa, como o tamanho do nome, o nome, o CPF e a idade, e os imprime no console.

Este código é um exemplo simples de um servidor de saída que aguarda a conexão de um cliente e manipula os dados recebidos do cliente de acordo com as instruções definidas no método `run`.

```
You, 3 seconds ago | 1 author (You)
class Connection extends Thread {
    DataInputStream in;
    DataOutputStream out;
    Socket clientSocket;

    public Connection(Socket aClientSocket) {
        try {
            clientSocket = aClientSocket;
            in = new DataInputStream(clientSocket.getInputStream());
            out = new DataOutputStream(clientSocket.getOutputStream());
            this.start();
        } catch (IOException e) {
            System.out.println("Connection:" + e.getMessage());
        }
    }

    public void run() {
        try { // an echo server
              // Exemplo de vetor de pessoas
            Pessoa pessoa1 = new Pessoa(nome:"Joao", cpf:"123456789", idade:30);
            Pessoa pessoa2 = new Pessoa(nome:"Maria", cpf:"987654321", idade:25);
            Pessoa[] pessoas = { pessoa1, pessoa2 };

            // Envia o número de pessoas
            this.out.writeInt(pessoas.length);
            // Envia as pessoas      You, now • Uncommitted changes
            for (int i = 0; i < pessoas.length; i += 1) {
                this.out.writeInt(pessoas[i].getNome().getBytes(charsetName:"UTF-8").length);
                this.out.write(pessoas[i].getNome().getBytes());
                this.out.writeUTF(pessoas[i].getCpf());
                this.out.writeInt(pessoas[i].getIdade());
            }
        } catch (EOFException e) {
            System.out.println("EOF:" + e.getMessage());
        } catch (IOException e) {
            System.out.println("readline:" + e.getMessage());
        } finally {
            try {
                clientSocket.close();
```

Questão 3:

Nesta questão há duas classes auxiliadoras ou TAD's, sendo elas 'Mensagem' e 'Produto', onde ambas são utilizadas para guardar métodos para o tratamento desses TAD's, não havendo necessidade de explicá-las neste relatório, dito isso, passamos pras Classes 'Cliente' e 'Servidor'.

Cliente.java

O código apresentado é um cliente Java que se conecta a um servidor remoto e realiza várias operações de comunicação com ele. Aqui está uma análise detalhada do código:

- O código cria um `Socket` para se conectar a um servidor remoto com endereço IP e porta.

- Em seguida, cria um `BufferedReader` para ler as entradas do teclado e uma série de objetos de entrada e saída para a comunicação com o servidor.

- O cliente apresenta um menu de opções que permite ao usuário escolher entre listar produtos, trocar produto, cadastrar produto, remover produto ou sair.

- Cada opção do menu é implementada usando um `switch case` e envolve a troca de diferentes tipos de mensagens com o servidor. Essas trocas se dão por meio de empacotamento e desempacotamento de cada requisição do cliente e cada resposta do servidor.

- Cada troca de mensagem é tratada de forma específica, com solicitações e respostas adequadas sendo enviadas e recebidas entre o cliente e o servidor.

- O código é envolto em blocos `try-catch` e `finally` para lidar com possíveis exceções e garantir que os recursos sejam fechados corretamente, incluindo o próprio socket, bem como os fluxos de entrada e saída de objetos.

No geral, o código implementa um cliente que interage com um servidor remoto por meio de trocas de mensagens estruturadas, permitindo que o usuário execute operações específicas no servidor.

```java
switch (tipoProduto) {
    case "1":
        // Cliente deseja trocar um livro
        // Empacotar a requisição de troca de produto
        oos = new ObjectOutputStream(socket.getOutputStream());
        mensagem = new Mensagem(conteudo:"troca");
        oos.writeObject(mensagem);

        // Desempacotar a resposta do servidor
        ois = new ObjectInputStream(socket.getInputStream());
        reply = (Mensagem) ois.readObject();
        // "Menu de troca:"
        System.out.println(reply.getConteudo());

        // Lendo o nome do produto que o cliente deseja trocar e as informações do novo
        // produto
        System.out.println(x:"Informe o nome do livro antigo: ");
        String nomeProdutoAntigo = br.readLine();
        System.out.println(x:"Informe o nome do livro novo: ");
        String nomeNovoProduto = br.readLine();
        System.out.println(x:"Informe o preço do livro novo: ");
        double precoNovoProduto = Double.parseDouble(br.readLine());
        System.out.println(x:"Informe o autor do livro novo: ");
        String autorNovoProduto = br.readLine();
        System.out.println(x:"Informe o número de páginas do livro novo: ");
        int numPaginasNovoProduto = Integer.parseInt(br.readLine());

        // Empacotando a mensagem
        oos = new ObjectOutputStream(socket.getOutputStream());
        mensagem = new Mensagem(tipoProduto + ", " + nomeProdutoAntigo + ", " + nomeNovoProduto
                + ", " + precoNovoProduto + ", " + autorNovoProduto + ", "
                + numPaginasNovoProduto);
        oos.writeObject(mensagem);

        // Desempacotando a mensagem de resposta do servidor
        // Conteudo: "Troca realizada com sucesso!"
        ois = new ObjectInputStream(socket.getInputStream());
        reply = (Mensagem) ois.readObject();
        System.out.println(reply.getConteudo());
```

Servidor.java

O código é o servidor correspondente ao cliente discutido anteriormente. Ele lida com as solicitações do cliente e executa as operações necessárias em uma estrutura de dados de controle. Aqui está uma análise detalhada do código:

- O código começa definindo a porta do servidor e inicializando um `ServerSocket` nessa porta.
- Em seguida, cria um objeto `Controle` que gerenciará as operações do servidor.
- O servidor inicia e espera a conexão de um cliente. Em seguida, entra em um loop infinito para receber mensagens do cliente e responder adequadamente.

- As mensagens do cliente são desempacotadas e tratadas de acordo com as operações solicitadas, como troca, cadastro, remoção ou listagem de produtos.

- Para cada operação, o servidor interage com o objeto `Controle` para executar as ações necessárias nos produtos.

- O servidor empacota e envia mensagens de resposta de volta ao cliente para informar o status da operação solicitada.

- O código envolve a lógica do servidor em blocos `try-catch` para lidar com possíveis exceções, além de garantir que os recursos sejam fechados corretamente, incluindo o socket do servidor e os fluxos de entrada e saída de objetos.

No geral, o servidor implementa a lógica para lidar com as solicitações do cliente e manipular as operações correspondentes nos produtos de acordo com as informações recebidas do cliente.

```java
try {

    // Iniciando o servidor
    System.out.println("Servidor iniciado na porta " + PORTA);      You, 5 hours ago • final update
    socket = serverSocket.accept();
    System.out.println("Cliente conectado: " + socket.getInetAddress().getHostAddress());

    // Loop infinito para receber as mensagens do cliente
    while (true) {
        // Desempacotando a mensagem recebida sobre qual operação o cliente deseja
        ois = new ObjectInputStream(socket.getInputStream());
        mensagem = (Mensagem) ois.readObject();

        System.out.println("Mensagem recebida: " + mensagem.getConteudo());

        // Empacotando a mensagem de resposta do servidor a depender da operação
        // Caso o cliente deseje Trocar um produto
        if (mensagem.getConteudo().equals(anObject:"troca")) {
            // Empacotando a mensagem de resposta do servidor
            oos = new ObjectOutputStream(socket.getOutputStream());
            reply = new Mensagem(
                    conteudo:"Menu de Troca:");
            oos.writeObject(reply);

            // Desempacotando a mensagem recebida
            // Conteudo: tipoProduto, nomeProdutoAntigo, nomeProdutoNovo, precoProdutoNovo,
            // autorProdutoNovo, numPaginasProdutoNovo
            // Que serão agrupadas em um array de Strings:
            // info1, info2, ...
            ois = new ObjectInputStream(socket.getInputStream());
            mensagem = (Mensagem) ois.readObject();

            // Qual o tipo do produto?
            String[] info = mensagem.getConteudo().split(regex:", ");
            String tipoProduto = info[0];

            for (int i = 0; i < info.length; i++) {
                System.out.println(info[i]);
            }
```

Questão 4:

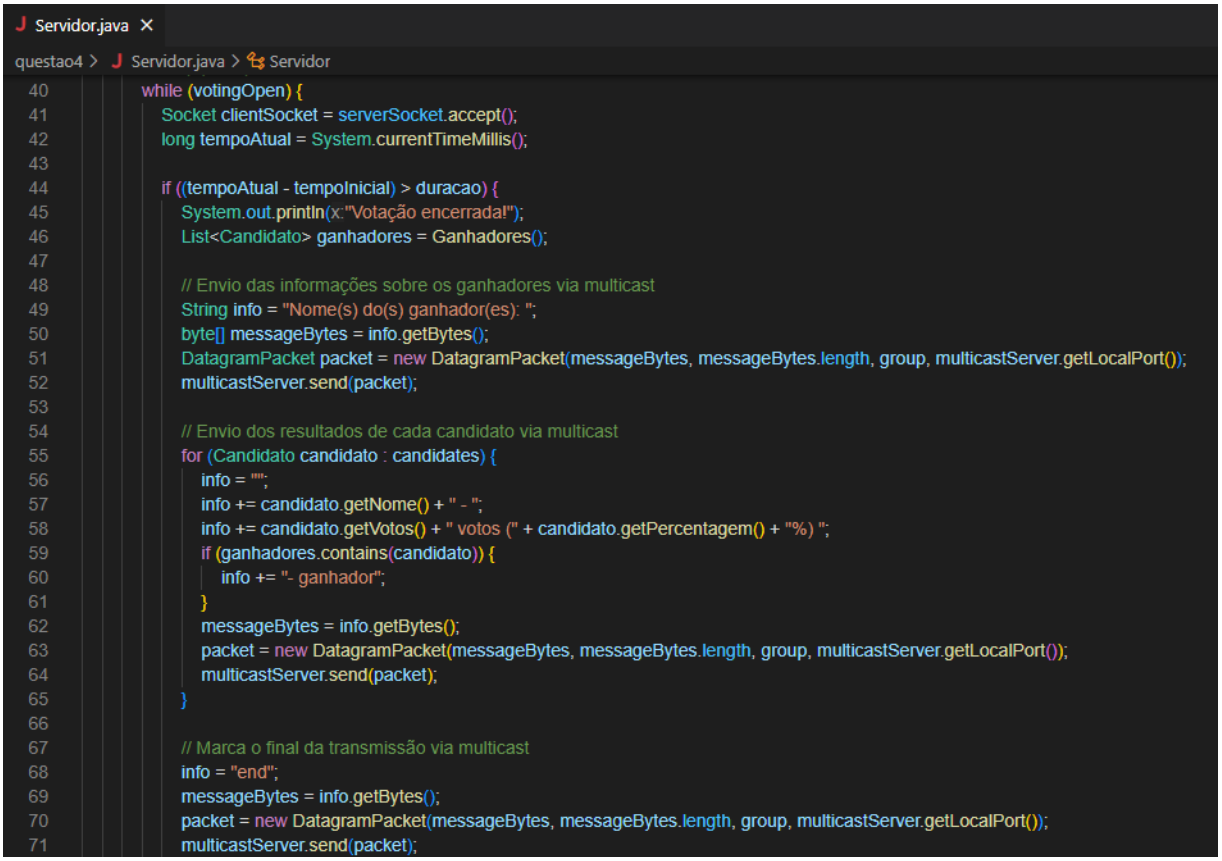
Nesta questão há duas classes auxiliadoras ou TAD's, sendo elas 'Candidato' e 'Voto', onde ambas são utilizadas para guardar métodos para o tratamento desses TAD's, não havendo necessidade de explicá-las neste relatório, dito isso, passamos pras Classes 'Cliente' e 'Servidor'.

Servidor.java

O código representa um servidor de votação em Java. O servidor é projetado para gerenciar uma votação com candidatos, eleitores e a capacidade de administradores interagirem com o sistema para adicionar ou remover candidatos e enviar informes. A votação é controlada por um servidor TCP, e os resultados são transmitidos aos clientes por meio de um servidor UDP multicast. Aqui está uma análise detalhada do código:

- O servidor abre um servidor TCP para aguardar conexões de clientes (eleitores e administradores) e configura um servidor UDP multicast para enviar resultados e informes.
- Eleitores podem ver a lista de candidatos e votar em um deles.
- Administradores com CPF "admin" têm acesso a funcionalidades adicionais, como adicionar e remover candidatos, exibir informações sobre os candidatos e enviar informações gerais para os eleitores.
- A votação é controlada pelo tempo, com um limite de duração configurado pelo administrador.
- Após o encerramento da votação, os resultados são enviados aos clientes via multicast, incluindo os nomes dos candidatos vencedores e suas porcentagens de votos.

No geral, o código implementa um servidor que permite a votação de candidatos e a administração da eleição por administradores. Ele demonstra o uso de sockets TCP e UDP multicast para comunicação e transmissão de resultados.



Cliente.java

O código do Cliente em Java faz parte de um sistema de votação e interação com um servidor. Aqui está uma análise detalhada do código:

- O código define as constantes SERVER_ADDRESS, UDP_SERVER_ADDRESS, SERVER_PORT_TCP, e SERVER_PORT_UDP para especificar os endereços e portas do servidor e da comunicação UDP.
- Também cria um MulticastSocket para comunicação multicast.
- A função principal inicia uma conexão com o servidor via TCP, lê o CPF do cliente e determina se o cliente é um "admin" ou eleitor.
- Para eleitores, inicia a comunicação multicast e chama a função eleitor, que implementa a lógica de votação.
- Para "admin", chama a função administrador, que lida com as ações administrativas, como mostrar a lista de candidatos, inserir ou remover candidatos, enviar informativos e sair..
- Para eleitor, solicita a lista de candidatos ao servidor, permite que o eleitor vote em um candidato informando o número e aguarda informações do servidor.
- Usa um loop para manter a interação com o administrador até que ele decida sair.
- LerCandidatos: Solicita e exibe a lista de candidatos.
- InserirCandidato: Solicita o nome de um novo candidato, envia ao servidor e exibe uma confirmação.
- RemoverCandidato: Solicita a remoção de um candidato, envia ao servidor e exibe uma confirmação.

- EnviarInformativos: Solicita e envia informativos ao servidor.
- Recebe informativos multicast do servidor até que a mensagem "end" seja recebida, indicando o fim dos informativos.

No geral, o código implementa um sistema para que eleitores e administradores possam interagir com o servidor de votação. Ele utiliza comunicação TCP para interações individuais e comunicação multicast para distribuir informações a todos os clientes (caso seja admin) e para receber informações (caso seja um eleitor).