1. Arquitetura do Sistema

Arquitetura Adotada: Cliente-Servidor

O sistema foi modelado utilizando a arquitetura Cliente-Servidor, uma abordagem fundamental em sistemas distribuídos.

- **Nó Servidor (Server):** Existe um único processo servidor que atua como autoridade central. Ele é responsável por gerenciar o estado da aplicação (se a votação está aberta ou fechada) e por armazenar os dados de forma consistente (a contagem de votos).
- **Nós Clientes (Clients):** Múltiplos processos clientes podem se conectar ao servidor. Cada cliente representa um terminal de votação, responsável por interagir com o eleitor, coletar seu voto e enviá-lo ao servidor para processamento.

Por que essa arquitetura foi escolhida?

- **Centralização do Controle:** Para um sistema de eleição, é crucial que a contagem de votos e o status da votação (aberta/fechada) sejam gerenciados em um único local para garantir consistência e evitar discrepâncias.
- **Simplicidade e Clareza:** O modelo Cliente-Servidor é intuitivo e fácil de implementar e explicar, sendo ideal para os requisitos do projeto.
- **Escalabilidade:** É fácil adicionar mais clientes (nós) ao sistema sem precisar modificar a lógica do servidor. O servidor foi projetado com um *pool de threads* para lidar com várias conexões de clientes simultaneamente.

---

2. Ferramentas e Tecnologias Utilizadas

- **Linguagem de Programação: Java**
  - **Motivo:** Java possui um suporte nativo robusto e maduro para programação de rede (com o pacote java.net) e para concorrência (Threads), que são os pilares deste projeto.
- **IDE:** VS Code.
- **Comunicação em Rede: API de Sockets Java**.

---

3. Escolha da Tecnologia: Socket vs. RMI

Por que Sockets em vez de RMI?

1. **Controle e Transparência:** Sockets nos dão controle total sobre os dados que são enviados pela rede. Para fins didáticos, isso é excelente, pois permite demonstrar *explicitamente* como as mensagens são criadas, enviadas, recebidas e interpretadas, tornando o conceito de comunicação em rede mais tangível.

2. **Adequado à Simplicidade do Problema:** Nossa aplicação troca mensagens de texto simples. A complexidade adicional do RMI, que permite invocar métodos em objetos remotos como se fossem locais, seria um exagero para essa tarefa e ocultaria os detalhes da comunicação que queríamos demonstrar.
3. **Flexibilidade:** O protocolo baseado em texto é simples e poderia ser facilmente implementado por clientes em outras linguagens de programação, o que não seria tão direto com RMI.

---

4. Funções e Métodos Principais de Cada Classe

a) `VotingServer.java`

- **Função:** É o coração do sistema. Ele inicializa o serviço, aceita conexões de clientes e gerencia o estado e os dados da votação.
- **Principais Métodos:**
  - `main()`: Ponto de entrada que cria e inicia a instância do servidor.
  - `startServer()`: Cria o `ServerSocket` na porta `12345`, gerencia um *pool de threads* para os clientes e entra em um loop infinito aceitando novas conexões (`serverSocket.accept()`). Para cada nova conexão, ele delega o trabalho para um `ClientHandler`.
  - `registerVote(int candidateNumber)`: Método `synchronized` para garantir que apenas uma thread modifique a contagem de votos por vez (evitando *race conditions*). Ele verifica se a votação está aberta e registra o voto.
  - `setVotingOpen(boolean isOpen)`: Permite que o administrador (via `ServerAdminConsole`) altere o estado da votação, implementando o controle de acesso.

b) `ClientHandler.java`

- **Função:** É um "assistente" do servidor. Cada instância dessa classe roda em sua própria thread e é responsável por se comunicar com um único cliente.
- **Principais Métodos:**
  - `run()`: Lógica principal da thread. Lê a mensagem de identificação do cliente, e depois entra em um loop para ler as mensagens de voto. Chama os métodos do `VotingServer` para processar as solicitações e envia as respostas de volta ao cliente.
  - `processMessage(String message)`: Interpreta as mensagens recebidas do cliente (ex: `VOTE:SYNC:20`) e aciona as ações correspondentes.

c) `ServerAdminConsole.java`

- **Função:** Roda em uma thread separada no servidor e fornece uma interface de linha de comando para um administrador gerenciar a eleição.
- **Principais Métedos:**
    - `run()`: Entra em um loop infinito que lê comandos do console (`abrir`, `fechar`, `resultados`).
    - `printResults()`: Acessa o mapa de votos do `VotingServer` e exibe a contagem atual.

d) `VotingClient.java`

- **Função:** É a interface do eleitor. Roda como um programa independente que se comporta como um terminal de votação.
- **Principais Métodos:**
    - `main()`: Contém o loop principal do "terminal". A cada iteração, ele pede o nome de um eleitor, estabelece uma nova conexão com o servidor (`new Socket(...)`), envia o voto e fecha a conexão automaticamente (graças ao bloco `try-with-resources`).
    - `printVoteMenu(String name)`: Exibe as opções de voto para o eleitor.

---

5. Fluxo de Comunicação (Exemplo de um Voto)

1. **Início:** O `VotingServer` é executado. Ele abre a porta 12345 e a `ServerAdminConsole` é iniciada. O administrador digita `abrir`.
2. **Conexão:** Um `VotingClient` é executado. Ele pede o nome ("Ana") e cria um `Socket` para `127.0.0.1:12345`.
3. **Delegação:** No servidor, `serverSocket.accept()` retorna um novo socket para essa conexão. O `VotingServer` cria uma nova instância de `ClientHandler` com esse socket e a executa em uma nova thread do pool.
4. **Identificação:** O Cliente envia a mensagem de texto `"IDENTIFY:Ana"`. O `ClientHandler` correspondente lê essa mensagem, armazena o nome "Ana" e imprime no console do servidor que "Ana" se conectou.
5. **Ação do Usuário:** O Cliente exibe o menu. "Ana" escolhe a opção 1 (Síncrono) e digita o candidato `20`.
6. **Envio da Mensagem:** O Cliente formata e envia a mensagem `"VOTE:SYNC:20"` através do stream de saída do seu socket.
7. **Processamento:** O `ClientHandler` de "Ana" recebe a mensagem. Ele a interpreta e chama o método `server.registerVote(20)`.
8. **Lógica Central:** O `VotingServer` (de forma *thread-safe*) verifica que a votação está aberta, incrementa a contagem do candidato 20 e retorna a string `"VOTO_CONFIRMADO:..."`.
9. **Resposta:** O `ClientHandler` recebe essa string de volta e a envia para o Cliente através do stream de saída.

10. **Finalização:** O `VotingClient` recebe a confirmação e a exibe para "Ana". O bloco `try-with-resources` do cliente termina, fechando automaticamente o socket. Do lado do servidor, o `ClientHandler` detecta que a conexão foi fechada e sua thread é encerrada e devolvida ao pool.
11. **Próximo Eleitor:** O loop principal do `VotingClient` recomeça, pronto para um novo eleitor.