

**Instituto Federal de Educação, Ciência e Tecnologia da Bahia**

**Campus**: Santo Antônio de Jesus

**Data**: 10/05/2025

**Curso**: Analise e Desenvolvimento de Sistemas

**Disciplina**: Sistemas Distribuídos

**Docente**: Felipe Silva

**Discentes:** Anderson Guilherme Santana Moraes e Carlos de Jesus Conceição Junior

**Relatório Técnico: Sistema de Pedidos e**

**Pagamentos para Quiosques Autônomos**

**Sumário**

1. Introdução

2. Escolha e Justificativa do Modelo Arquitetural

3. Definição da Comunicação Remota

4. Diagrama Arquitetural

5. Explicação do Funcionamento da Comunicação

6. Protocolos, Bibliotecas e Frameworks Utilizados

7. Comparação Detalhada: Sockets vs. RMI

8. Desafios Enfrentados e Soluções

9. Prints e Logs de Testes Realizados

10. Referências Utilizadas

**Introdução**

Este projeto é uma simulação de um sistema de pedidos para uma praça de alimentação, desenvolvido como uma aplicação prática para a disciplina. Ele consiste em um servidor central, apresentado como um "Painel de Controle", que gerencia e exibe os pedidos em tempo real, e múltiplos clientes, que são as janelas de "Quiosque" interativas.

O grande diferencial do projeto é que ele foi construído com duas tecnologias de comunicação distintas para fins de comparação:

1. **Sockets TCP:** Uma abordagem de baixo nível, onde a comunicação é feita manualmente através de um protocolo de texto customizado, simulando uma "ligação direta" entre o quiosque e o servidor.
2. **Java RMI:** Uma abordagem de alto nível, onde o quiosque simplesmente "chama um método" do servidor como se fosse local, e a tecnologia RMI se encarrega de toda a complexidade da rede.

O resultado são dois sistemas que, para o usuário, têm uma aparência e funcionalidade idênticas, mas operam de maneiras fundamentalmente diferentes, demonstrando na prática os conceitos de arquitetura e comunicação distribuída.

**1. Escolha e Justificativa do Modelo Arquitetural**

* **Modelo Escolhido:** Cliente-Servidor.
* **Justificativa Técnica:** A arquitetura Cliente-Servidor foi selecionada por ser a mais adequada e eficiente para o desafio de um sistema de pedidos para quiosques autônomos. As Justificativas para a escolha:
  + **Centralização e Consistência:** Um servidor central permite gerenciar de forma unificada o cardápio, o recebimento de pedidos e o processamento de pagamentos. Isso garante que todos os clientes (quiosques) operem com as mesmas informações.
  + **Escalabilidade:** O modelo facilita a adição de novos quiosques à praça de alimentação. Cada novo quiosque torna-se apenas um cliente se conectando ao servidor, sem a necessidade de reconfigurar os clientes existentes.
  + **Separação de Responsabilidades:** A lógica de negócio principal reside no servidor, simplificando o desenvolvimento e a manutenção dos clientes. Os quiosques precisam apenas se preocupar com a interface do usuário e o envio de solicitações.
  + **Baixa Latência em Rede Local:** Para o ambiente proposto (uma praça de alimentação), a comunicação ocorre em uma rede local (LAN). A latência da comunicação direta entre cliente e servidor é mínima e perfeitamente adequada para a rápida resposta exigida ao fazer um pedido.
  + **Simplicidade e Gerenciamento:** Comparado a modelos como P2P ou Microservices, o Cliente-Servidor é mais simples de implementar e gerenciar para o escopo deste projeto, permitindo focar nos mecanismos de comunicação remota. 1

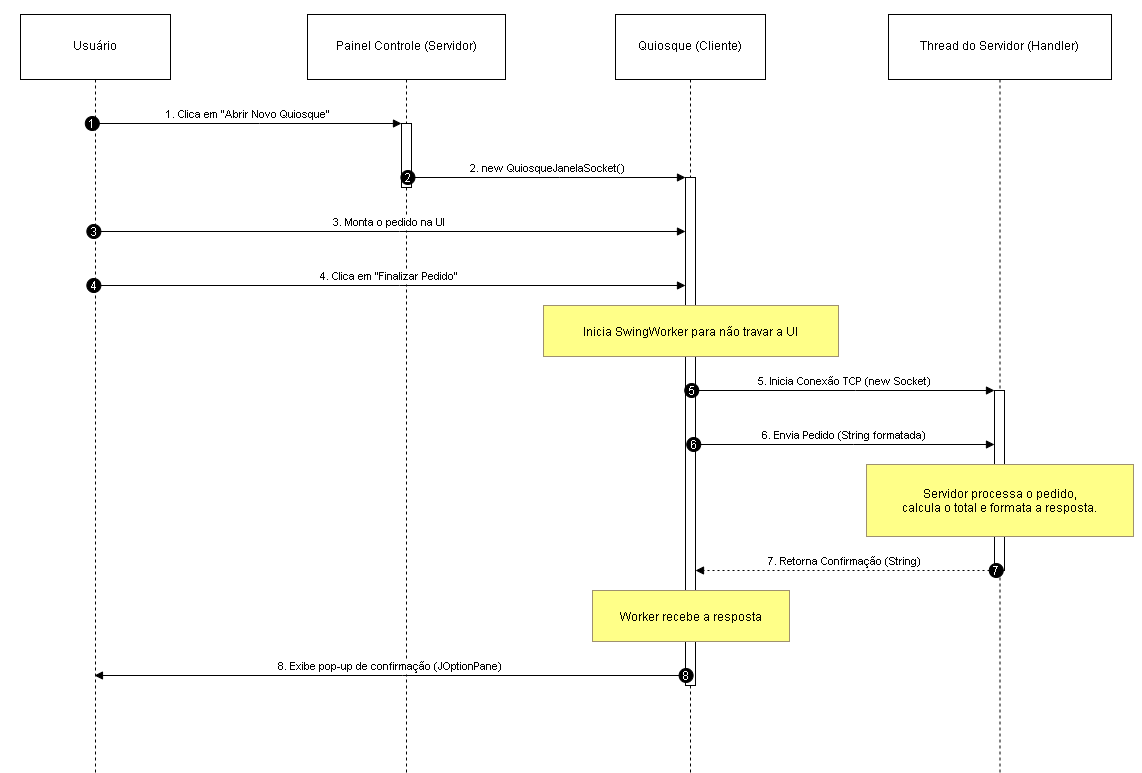
**2. Definição da Comunicação Remota**

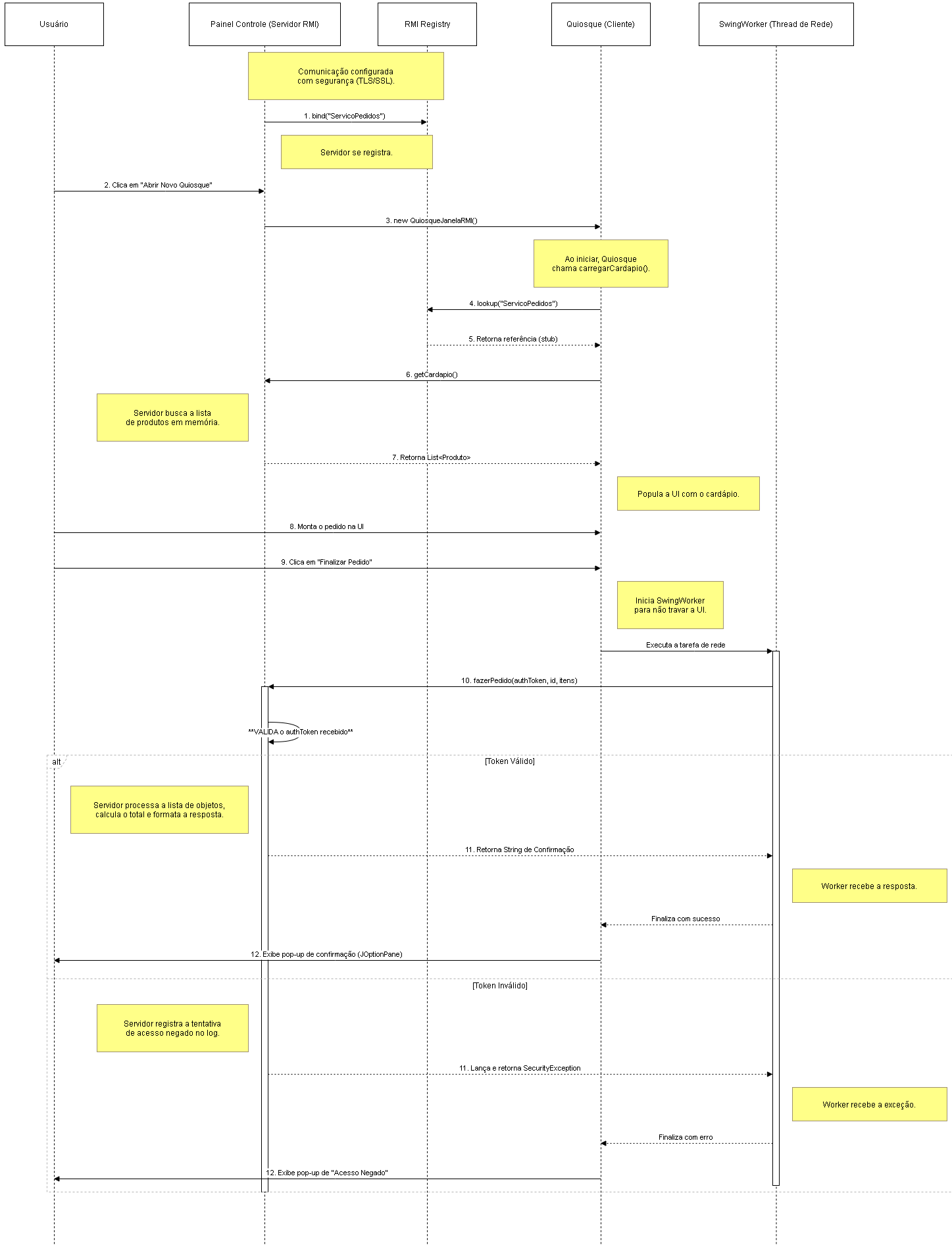
Para cumprir os objetivos da atividade e explorar diferentes paradigmas de comunicação, o sistema foi implementado em duas versões distintas.

* **Escolha 1: Sockets TCP**
  + **Motivo:** A escolha de Sockets TCP se deu pela necessidade de uma comunicação de baixo nível, confiável e orientada à conexão. Sockets garantem que os dados dos pedidos cheguem ao servidor de forma íntegra e na ordem correta, o que é crítico para uma aplicação de pedidos.
* **Escolha 2: Java RMI (Remote Procedure Call)**
  + **Motivo:** Java RMI foi escolhido como representante de uma tecnologia de RPC. O objetivo do RMI é abstrair a complexidade da rede, permitindo que um objeto cliente invoque métodos em um objeto servidor como se fossem chamadas locais. Essa abordagem aumenta a produtividade do desenvolvedor e demonstra um paradigma de comunicação de mais alto nível.

**3. Diagrama Arquitetural**

**Arquitetura baseada em Sockets**

****

**Arquitetura baseada em RMI**

**4. Explicação do Funcionamento da Comunicação**

**Implementação com Sockets**

* **Servidor:** A classe PainelControleSocket inicia um ServerSocket na porta 12345 em uma thread separada para não bloquear a interface gráfica. Para cada cliente que se conecta, uma nova thread ClientHandler é instanciada. Essa thread fica responsável por receber os dados, processar o pedido e enviar uma resposta, garantindo que o servidor possa atender múltiplos quiosques concorrentemente.
* **Cliente:** A classe QuiosqueJanelaSocket apresenta a interface gráfica ao usuário. O cardápio é carregado de uma lista fixa localmente.
* **Comunicação:** Ao clicar em "Finalizar Pedido", o cliente formata o carrinho de compras em uma única string, seguindo um protocolo customizado (ID:%d;PEDIDO:%s). Esta string é enviada de uma vez ao servidor através de um Socket. Um SwingWorker é utilizado para realizar a comunicação em uma thread de segundo plano, mantendo a UI responsiva.
* **Processamento no Servidor:** O ClientHandler no servidor recebe a string, a divide (split) para extrair o ID do quiosque e os itens do pedido, calcula o valor total consultando sua própria lista de produtos e, por fim, envia uma string de confirmação de volta ao cliente pela mesma conexão.

**Implementação com RMI**

* **Servidor:** O PainelControleRMI estende UnicastRemoteObject e implementa a interface remota InterfacePedidos. Ao iniciar, ele se registra no *RMI Registry* na porta 1099, tornando seus métodos getCardapio() e fazerPedido() disponíveis para invocação remota. A comunicação é protegida usando SslRmiClientFactory e SslRmiServerFactory.
* **Cliente:** Ao ser iniciada, a QuiosqueJanelaRMI primeiro invoca o método remoto getCardapio() para popular dinamicamente sua lista de produtos, buscando-a diretamente do servidor.
* **Comunicação**: O usuário monta o pedido, que é armazenado como uma List<ItemPedido>. Ao finalizar, o cliente invoca o método remoto fazerPedido(authToken, kioskId, carrinho), passando um token de autenticação, seu ID e a lista de objetos do pedido diretamente. Toda a complexidade de serializar a lista de objetos, enviá-la pela rede de forma segura (TLS) e entregá-la ao servidor é gerenciada pela infraestrutura do RMI.
* **Processamento no Servidor:** O servidor primeiro valida o authToken recebido contra uma lista de tokens válidos (validTokens). Se o token for válido, o servidor recebe a List<ItemPedido> já desserializada, processa os itens para calcular o total e retorna uma string de confirmação. Se o token for inválido, uma SecurityException é lançada.

**Modelagem da Funcionalidade de Pagamento**

A funcionalidade de "pagamentos" foi implementada de forma simulada, focando na lógica da transação dentro da nossa arquitetura, conforme detalhado abaixo:

* **Lógica Centralizada no Servidor:** Toda a responsabilidade de processar a transação foi atribuída ao servidor. Isso garante que as regras de negócio, como preços e cálculos, sejam consistentes para todos os quiosques.
* **Servidor como Autorizador:** Ao receber um pedido, o servidor executa as seguintes ações que simulam uma autorização de pagamento:
  + Valida os itens recebidos.
  + Calcula o valor total da compra com base nos preços do seu cardápio.
  + Gera um número de pedido único para a transação.

**Protocolos, Bibliotecas e Frameworks Utilizados**

A construção do projeto se baseou em bibliotecas nativas do Java (JDK), organizadas pelas suas funcionalidades principais dentro do sistema:

**Comunicação em Rede e Distribuída**

* **java.net.\***: Base da implementação com Sockets. As classes ServerSocket e Socket foram usadas para estabelecer a comunicação TCP/IP entre o PainelControleSocket e os QuiosqueJanelaSocket, permitindo a troca de mensagens baseada em streams de texto.
* **java.rmi.\***: Essencial para a implementação RMI. Foram utilizadas as interfaces e classes como Remote, LocateRegistry para o registro de serviços, e UnicastRemoteObject para criar o objeto remoto que implementa a InterfacePedidos.

**Segurança**

* **javax.net.ssl.\***: Utilizada para implementar a camada de segurança (TLS) sobre a comunicação RMI. Classes como SSLContext, KeyManagerFactory e TrustManagerFactory foram cruciais para configurar a comunicação criptografada, garantindo a confidencialidade e a integridade dos dados trocados.
* **java.security.\***: Usada para o gerenciamento de chaves e certificados. A classe KeyStore foi empregada para carregar o arquivo server.keystore, que contém o certificado do servidor necessário para a comunicação SSL. A autenticação dos pedidos na versão RMI é feita através de um token.

**Interface Gráfica (GUI)**

* **javax.swing.\*** e **java.awt.\***: Foram as bibliotecas escolhidas para a construção de toda a interface gráfica do sistema. Componentes como JFrame, JButton, JTextArea, JComboBox e JScrollPane, junto com gerenciadores de layout como BorderLayout e FlowLayout, foram usados para criar tanto a janela do Painel de Controle quanto a dos Quiosques.

**Sincronização e Concorrência**

* **java.util.concurrent.atomic.AtomicInteger**: Utilizada no servidor para gerar os números de identificação dos pedidos (orderIdGenerator) de forma atômica. Isso garante que, mesmo com múltiplos quiosques fazendo pedidos simultaneamente, cada pedido receberá um ID único, evitando condições de corrida.

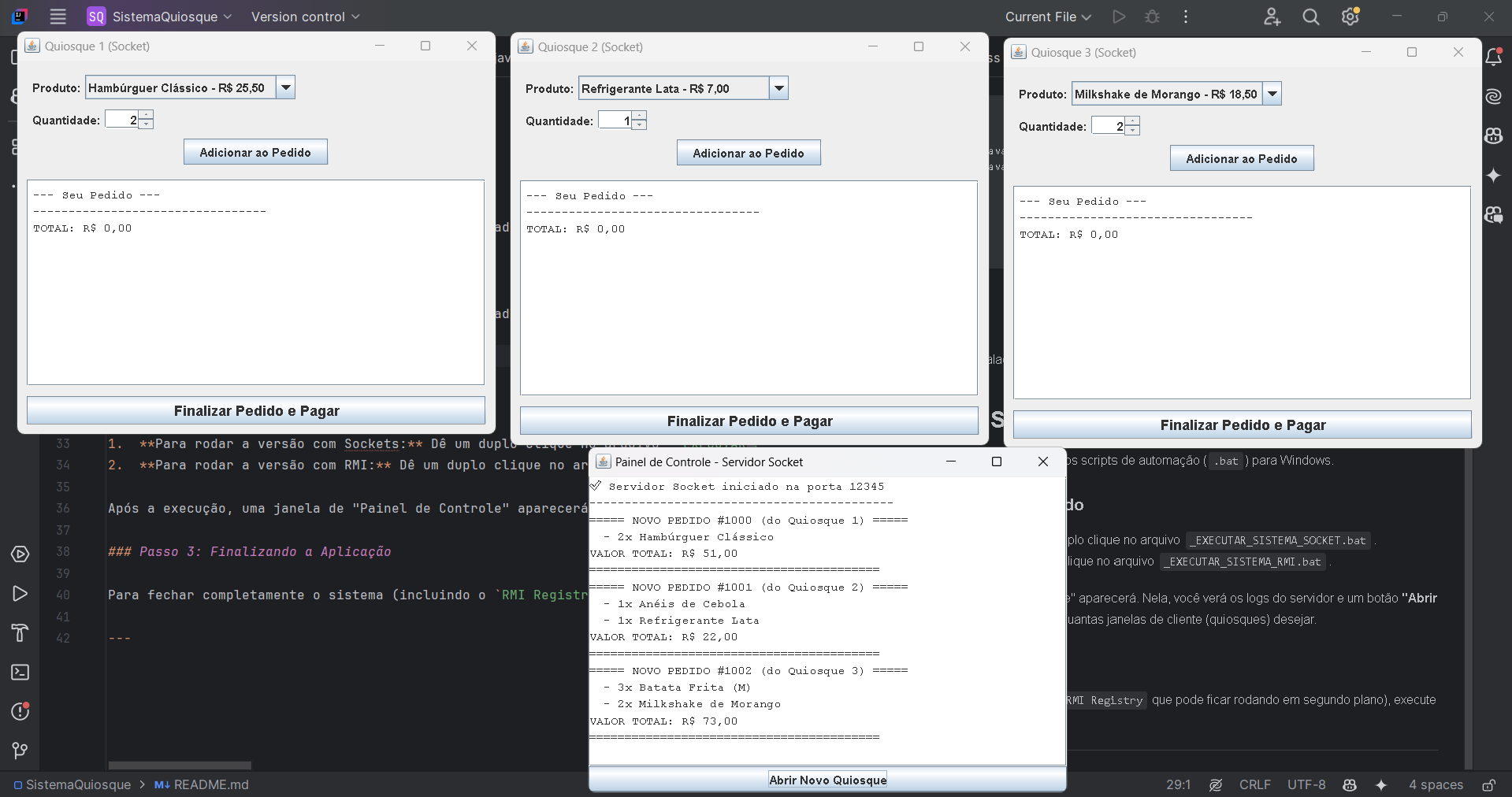
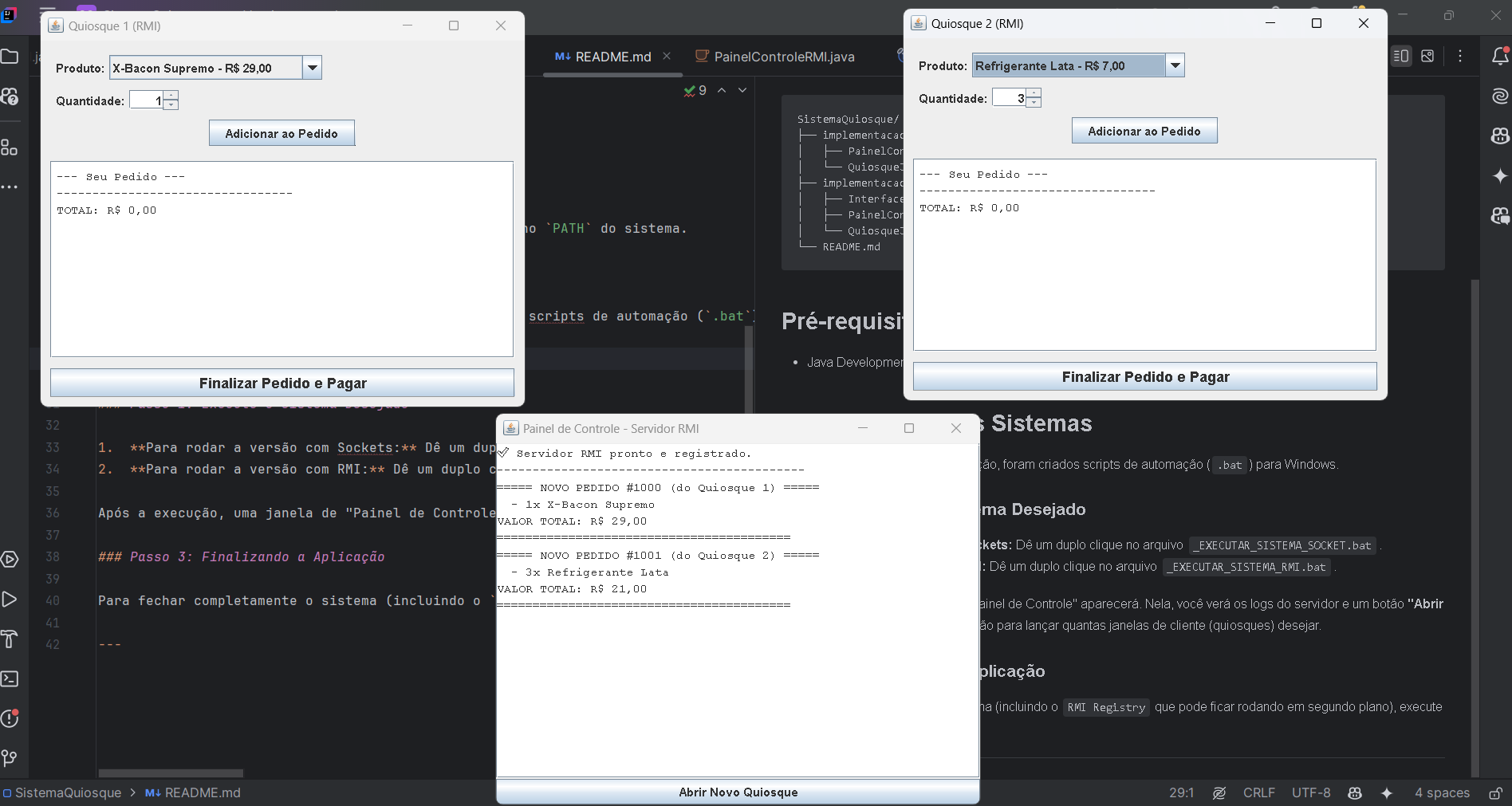
**5. Comparação: Sockets vs. RMI**

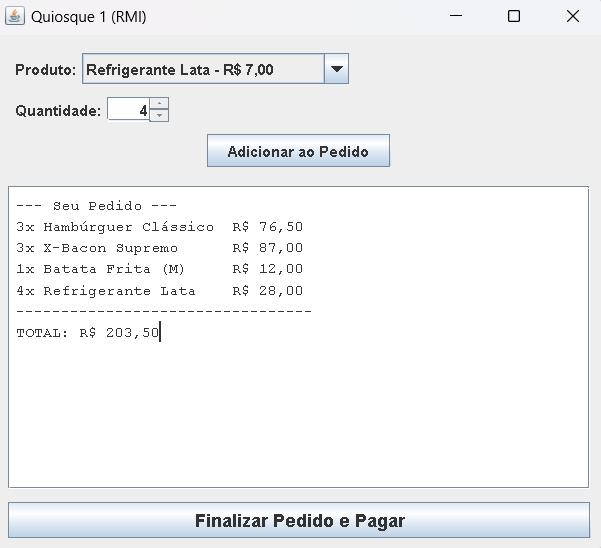
|  |  |  |
| --- | --- | --- |
| **Característica** | **Sockets TCP** | **Java RMI (RPC)** |
| **Nível de Abstração** | **Baixo.** Controle total sobre o fluxo de bytes e o protocolo da aplicação. | **Alto.** A rede é quase transparente. Foco na chamada de métodos em objetos remotos. |
| **Facilidade de Uso** | **Mais complexo.** Exige gerenciamento manual de conexões, threads e um protocolo de mensagens customizado. | **Muito mais simples.** A complexidade da rede é encapsulada, incluindo a serialização de objetos. |
| **Tratamento de Dados** | **Manual.** Foi necessário criar um protocolo de String customizado e fazer a serialização/parsing manual da lista de pedidos. | **Automático.** O RMI gerenciou nativamente a serialização da List<ItemPedido> e seus objetos Produto. |
| **Flexibilidade** | **Total.** Interoperável com qualquer sistema que se comunique via TCP/IP. | **Limitado.** Otimizado para comunicação entre aplicações Java. |
| **Tratamento de Erros** | Lida com exceções de baixo nível, como IOException. | Lida com RemoteException, que encapsula diversas falhas de rede. |
| **Segurança** | **Não implementada.** Exigiria uma implementação manual de TLS/SSL sobre o socket. | **Implementada.** O uso de SslRmiClientFactory e SslRmiServerFactory adicionou criptografia TLS à comunicação. |

**6. Desafios Enfrentados e Soluções**

* **Desafio 1: Concorrência de Clientes**
  + **Problema:** O servidor precisava ser capaz de atender múltiplos quiosques simultaneamente sem que um bloqueasse o outro.
  + **Solução (Sockets):** Foi implementado o padrão "Thread-per-Client". Para cada nova conexão de cliente aceita pelo ServerSocket, o servidor instanciava uma nova thread (ClientHandler) dedicada exclusivamente a essa comunicação.
* **Desafio 2: Responsividade da Interface Gráfica (UI)**
  + **Problema:** Operações de rede (tanto em Sockets quanto em RMI) são bloqueantes e, se executadas na *Event Dispatch Thread (EDT)* do Swing, congelariam a interface do usuário.
  + **Solução:** Em ambas as implementações de cliente (QuiosqueJanelaSocket e QuiosqueJanelaRMI), a lógica de comunicação de rede foi encapsulada dentro de um javax.swing.SwingWorker. Isso moveu as operações de rede para uma thread de segundo plano, mantendo a UI sempre responsiva.
* **Desafio 3: Implementação de Segurança e Autenticação (RMI)**
  + **Problema:** A equipe enfrentou dificuldades na implementação de uma comunicação segura com criptografia TLS e um sistema de autenticação por tokens, pois eram conceitos novos. Isso gerou interferências iniciais e exceções de segurança durante os testes.
  + **Solução: O desafio foi superado em duas etapas:**
    1. **Criptografia (TLS/SSL):** Foram criadas as classes SslRmiClientFactory e SslRmiServerFactory. Essas classes foram configuradas para usar um server.keystore, garantindo que toda a comunicação RMI ocorresse sobre um canal criptografado. O cliente também foi configurado para confiar no certificado presente neste keystore.
    2. **Autenticação**: Foi adicionado um parâmetro authToken do tipo String ao método fazerPedido na interface remota. O cliente passou a gerar um token único baseado em seu ID, e o servidor, antes de processar qualquer pedido, valida este token contra uma lista de tokens pré-aprovados (validTokens), lançando uma SecurityException em caso de falha na autenticação.
* **Desafio 4: Layout da Interface Gráfica**
  + **Problema**: Os gerenciadores de layout padrão do Swing (como FlowLayout) se mostraram pouco previsíveis para a estrutura desejada,fazendo com que componentes não fossem renderizados corretamente quando a janela era redimensionada.
  + **Solução:** A estrutura da UI foi refeita utilizando uma combinação de BorderLayout como gerenciador principal e BoxLayout com painéis aninhados (JPanel). Isso criou um layout robusto e previsível que funciona de forma consistente em ambas as implementações.

**7. Prints e Logs de Testes Realizados**

* **Print 1: Painel de Controle Socket com 3 quiosques abertos e logs de pedidos.**
* **Print 2: Painel de Controle RMI com 2 quiosques e logs de pedidos.**
* **Print 3: Janela de um quiosque com os pedidos prontos para serem finalizados.**

****

* **Print 4: Janela de um quiosque após receber a confirmação do pedido.**

****

**8. Referências Utilizadas**

* Material de aula da disciplina de Sistemas Distribuídos.
* Oracle. **Java SE Documentation - java.net.Socket**. Disponível em: <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/net/Socket.html>.
* Oracle. **Java RMI Tutorial**. Disponível em: <https://docs.oracle.com/javase/tutorial/rmi/index.html>.
* Oracle. **Java SE Documentation - Concurrency in Swing**. Disponível em: <https://docs.oracle.com/javase/tutorial/uiswing/concurrency/index.html>.1