Trabalho Prático – Etapa 1: Serviço PIX

Eduarda Tessari Pereira – 00343210
Guilherme D'Ávila Pinheiro – 00342346
Maximus Borges da Rosa – 00342337
Laura Reis Müller – 00290707
Universidade Federal do Rio Grande do Sul

27 de outubro de 2025

Disciplina: INF01151 – Sistemas Operacionais II
Professor: Eder John Scheid

1 Introdução

O presente trabalho tem como objetivo implementar um serviço distribuído de transferência de valores entre clientes, simulando o funcionamento de um sistema de pagamento semelhante ao PIX. O sistema é composto por dois programas distintos, cliente e servidor, que se comunicam por meio do protocolo UDP (*User Datagram Protocol*), possibilitando o envio e recebimento de requisições de transferência de forma concorrente.

Nesta Etapa 1, o foco está na implementação das funcionalidades básicas do serviço, incluindo:

- Gerenciamento das requisições de transferência
- Atualização dos dados dos clientes
- Manutenção dos dados do banco
- Consistência do histórico de transações

Para isso, são utilizadas *threads* para o processamento paralelo das requisições e mecanismos de exclusão mútua baseados no modelo leitor-escritor, garantindo acesso controlado às estruturas de dados compartilhadas.

2 Estrutura Geral do Sistema

O sistema implementado segue uma arquitetura cliente-servidor, dividida em duas fases principais de operação:

2.1 Fase de Descoberta

Nesta fase inicial, os clientes utilizam mensagens de *broadcast* UDP para descobrir o servidor na rede. O servidor responde com mensagens *unicast*, confirmando sua presença e registrando o novo cliente em sua tabela interna.

2.2 Fase de Processamento

Após a descoberta, os clientes enviam requisições de transferência ao servidor. O servidor processa cada requisição de forma concorrente, utilizando uma *thread* dedicada, e envia mensagens de confirmação (ACK) ao cliente solicitante.

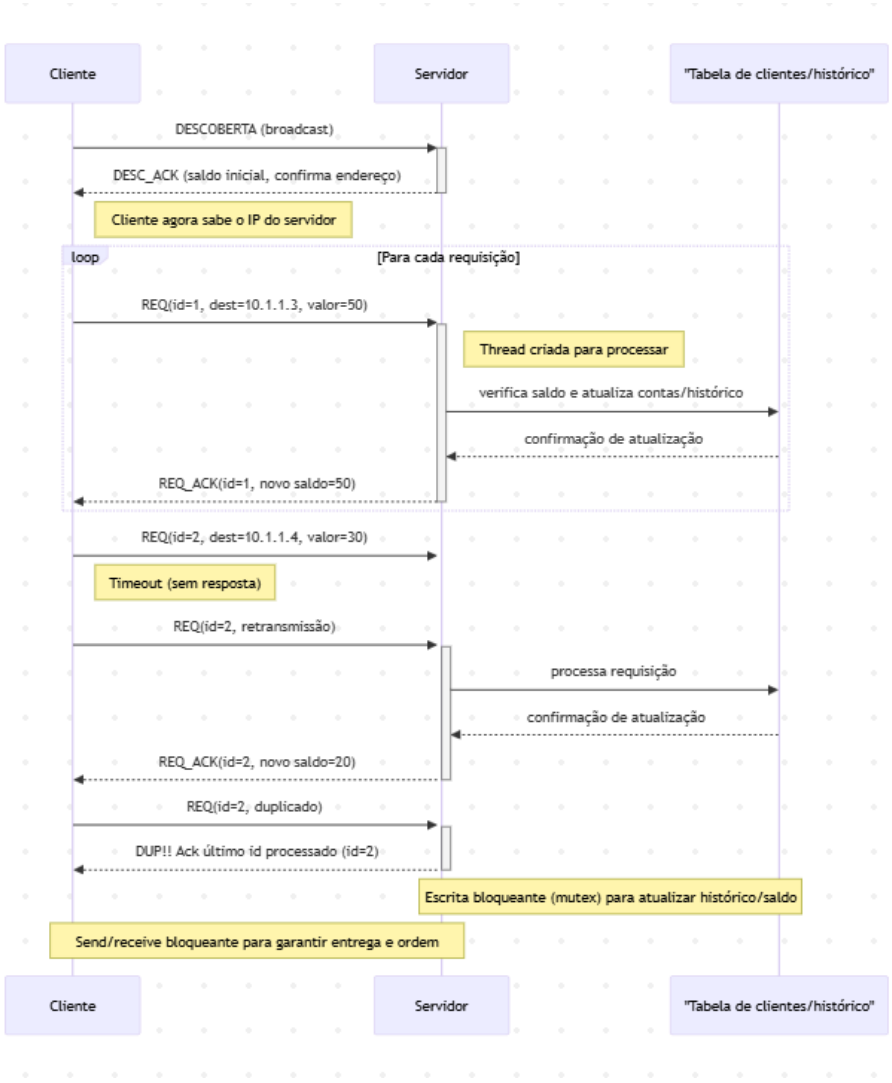

Figura 1: Diagrama cliente-servidor

3 Subserviços Implementados

3.1 Subserviço de Descoberta

O subserviço de descoberta foi implementado utilizando mensagens *broadcast* UDP na porta configurada. O processo funciona da seguinte forma:

1. O cliente cria um socket UDP e habilita a opção `SO_BROADCAST`
2. O cliente envia uma mensagem `PKT_DISCOVER` em *broadcast* (endereço 255.255.255.255)
3. O servidor recebe a mensagem, extrai o endereço IP do cliente e o registra no banco de dados
4. O servidor responde com uma mensagem `PKT_DISCOVER_ACK` via *unicast* diretamente ao cliente
5. O cliente aguarda a resposta com `select()` (timeout de 1 segundo)
6. Ao receber o ACK, o cliente armazena o endereço IP do servidor e encerra a fase de descoberta
7. Se não houver resposta, o cliente retenta até 50 vezes antes de desistir

Formato das mensagens:

```c
typedef struct {
    uint16_t type;    // Tipo do pacote (PKT_DISCOVER, PKT_DISCOVER_ACK, PKT_REQUEST, PKT_REQUEST_ACK)
    uint32_t seqn;    // Numero de sequencia da requisicao (ID no cliente)

    // Union permite que o pacote armazene diferentes estruturas no mesmo espaco
    union {
        RequestData req;
        AckData ack;
    };

} Packet;
```

```c
typedef struct {
    uint32_t seqn;         // Numero de sequencia do ACK
    uint32_t new_balance; // Novo saldo apos a transacao
    uint32_t dest_addr;   // IP do cliente destino
    uint32_t value;        // Valor transferido
    uint32_t server_addr; // IP do servidor
} AckData;
```

Estruturas utilizadas: O banco de dados do servidor (`server_db`) mantém uma tabela hash de clientes registrados, com as seguintes informações:

- **Chave:** Endereço IP do cliente (string)
- **Valor:** Estrutura contendo saldo, último `seqn` processado, e outras informações

3.2 Subserviço de Processamento

Envio de requisições:

O cliente implementa um componente responsável por enfileirar comandos vindos da interface e enviar cada requisição ao servidor de forma confiável usando retransmissões (RRA). Principais detalhes do funcionamento:

- **Fila thread-safe:** Comandos (tuplas destino, valor) são inseridos na fila por uma thread de leitura de entrada da interface via `enqueueCommand`. A fila é protegida por `mutex`.
- **Loop de processamento (`runProcessingLoop`):** Uma thread dedicada aguarda comandos na fila. Para cada comando ela constrói um `Packet` com `type = PKT_REQUEST`, depois chama a rotina bloqueante de envio com retransmissões. Só incrementa `_next_seqn` após receber com sucesso o `PKT_REQUEST_ACK` correspondente.
- **Mecanismo de retransmissão (RRA):** A função `sendRequestWithRetry` envia o pacote via `sendto` e aguarda ACK usando `select()` com timeout curto (configurado em milissegundos). Se o timeout expirar ou ocorrer erro, o cliente retransmite até um limite máximo de tentativas. Retransmissões e erros são registrados em log.
- **Validação de ACK:** Ao receber um pacote o cliente valida `type == PKT_REQUEST_ACK` e `seqn == request.seqn`. ACKs válidos são convertidos para `AckData` e repassados à thread de saída/interface através de `pushAck`. ACKs atrasados/duplicados ou pacotes inesperados são ignorados e logados.

Processamento no Servidor:

O servidor processa cada requisição de transferência de forma concorrente, criando uma nova thread para cada requisição recebida. O loop principal recebe um pacote, indentifica o tipo e chama o handler apropriado. Se a requisição é descoberta processa na thread principal, se é transação cria nova thread para processar, copiando os dados para eviitar condições de corrida. Saldos, histórico e estatísticas são atualizados de forma atômica.

A função `handleRequest` é responsável por validar, executar e confirmar cada requisição recebida do cliente. Em vez de apresentar o código completo aqui, descreve-se abaixo o comportamento e os pontos críticos da implementação:

1. **Controle de sequência:** Obtém-se `last_processed_seqn` do banco de dados e compara com o `packet.seqn` para classificar o pacote em:
   - **Duplicado:** `received_seqn <= last_processed_seqn` — reenvia-se o ACK bufferizado sem reprocessar.
   - **Fora de ordem:** `received_seqn > last_processed_seqn + 1` — descarta-se (ou reenvia o último ACK) para forçar retransmissão ordenada.
   - **Correto:** `received_seqn == last_processed_seqn + 1` — prossegue-se ao processamento.
2. **Processamento de Requisição:**
   - *Consulta de saldo* (`req.value == 0`): lê-se o saldo, atualiza-se `last_seqn` e armazena-se um ACK no buffer com o saldo atual. Não modifica estado do sistema, atualiza apenas `last_seqn`.

- *Transferência* (`req.value > 0`):
  - Utiliza estrutura `RequestData` contendo:
    - `dest_addr`: Endereço IP do destinatário (network order)
    - `value`: Valor da transferência em centavos
  - Valida saldo do cliente origem
  - Debita valor da conta origem
  - Credita valor na conta destino
  - Atualiza histórico de transações
  - Atualiza estatísticas globais
  - Retorna novo saldo em `ack.new_balance`
  - Atualiza `last_seqn` e buffer de ACK completo

3. **Envio de ACK:** No caso de duplicatas ou pacotes fora de ordem, o ACK reenviado corresponde ao último ACK válido.

4. **Notificação da interface**

3.3 Subserviço de Interface

A interface foi implementada com threads separadas para leitura (input) e exibição (output) tanto no cliente quanto no servidor, permitindo operação assíncrona e desacoplada da lógica de rede.

Cliente – Leitura de requisições (input):
Uma thread (`inputLoop`) lê linhas do `stdin` de forma bloqueante. Comandos válidos são empurrados para a fila thread-safe do componente de requisições. A fila usa `mutex` para sincronizar com a thread de processamento.

Cliente – Exibição de respostas (output):
Uma thread (`outputLoop`) aguarda ACKs (estrutura `AckData`) empurrados pela camada de envio (`ClientRequest`) via `pushAck`. Para cada ACK recebido ela converte endereços, adiciona timestamp e formata os dados. A separação em thread de output evita bloqueio do envio/retransmissão e garante respostas exibidas na ordem em que os ACKs chegam.

Servidor – Output de processamento:
O servidor possui um componente `ServerInterface` com uma thread dedicada que imprime logs de eventos e um resumo do banco. Na inicialização a thread emite um resumo inicial, depois, Handlers (por exemplo `ServerProcessing::handleRequest`) chamam `notifyUpdate` para enfileirar linhas de log com dados de requisições processadas que serão formatados e impressos na tela.

Exemplo de saída do cliente:

```
2025-10(...) server 172(...) id req 5 dest 10.0.0.2 value 5 new balance 75
2025-10(...) server 172(...) id req 6 dest 10.0.0.43 value 0 new balance 75
2025-10(...) server 172(...) id req 7 dest 10.0.0.2 value 5 new balance 70
```

Exemplo de saída do servidor:

```
2025-10(...) client 172(...) id_req 5 dest 10.0.0.2 value 5
num transactions 5 total transferred 25 total balance 200
2025-10(...) client 172(...) id_req 6 dest 10.0.0.43 value 0
num transactions 5 total transferred 25 total balance 200
2025-10(...) client 172(...) id_req 7 dest 10.0.0.2 value 5
num transactions 6 total transferred 30 total balance 200
```

4 Sincronização e Concorrência

4.1 Sincronização no Cliente

No lado do cliente, a concorrência surge a partir da divisão funcional entre as *threads* responsáveis pela leitura de comandos do usuário, pelo envio de requisições ao servidor e pela exibição das respostas recebidas (ACKs). Essas componentes compartilham estruturas de dados internas, como as filas de comandos e de confirmações, exigindo controle de acesso para evitar condições de corrida.

A fila de comandos, por exemplo, é preenchida pela `inputLoop()`, que lê instruções do usuário a partir da entrada padrão, e consumida pela `runProcessingLoop()`, responsável por empacotar e enviar as requisições UDP. Para impedir acessos simultâneos indevidos, essa estrutura é protegida por um `std::mutex`, em conjunto com uma variável de condição (`std::condition_variable`), que permite o bloqueio eficiente da *thread* consumidora até que um novo comando seja inserido.

De forma análoga, a fila de ACKs da interface do cliente segue o mesmo padrão de sincronização. Cada vez que um pacote de confirmação é recebido e validado por `sendRequestWithRetry()`, ele é repassado à interface por meio da função `pushAck()`, que utiliza bloqueio de exclusão mútua para inserir o ACK na fila e notificar a *thread* de saída.

Principais mecanismos utilizados:

- `std::mutex` para exclusão mútua no acesso às filas;
- `std::condition_variable` para coordenação entre produtor e consumidor;
- `lock_guard` e `unique_lock` para desbloqueio automático e seguro.

Assim, no cliente, a sincronização atua principalmente sobre filas internas de comunicação, assegurando que as interações entre as *threads* de entrada, envio e saída sejam ordenadas, seguras e livres de interferências.

4.2 Sincronização no Servidor

A complexidade de sincronização é maior no servidor, pois múltiplas *threads* processam simultaneamente requisições provenientes de diferentes clientes. Cada requisição pode envolver leitura e modificação de informações críticas, como saldos de contas, histórico de transações e sumário global do banco. Para garantir a consistência dessas estruturas sob alta concorrência, foi implementado um modelo de controle de acesso do tipo **leitor/escritor**, por meio da classe `RWLock` personalizada, construída sobre `pthread_mutex_t` e variáveis de condição POSIX.

```
class RWLock {
    pthread_mutex_t mutex;
    pthread_cond_t readers_cv;
    pthread_cond_t writers_cv;
    int readers_active;
    int writers_waiting;
    bool writer_active;
};
```

Principais operações:

- `read_lock()`: utilizada em consultas (ex.: `getClientBalance()`);
- `write_lock()`: empregada em atualizações e transferências (ex.: `makeTransaction()`);
- `ReadGuard`/`WriteGuard`: wrappers que garantem desbloqueio automático ao final do escopo.

O modelo leitor/escritor permite que diversas leituras ocorram em paralelo, por exemplo, consultas de saldo, desde que nenhuma escrita esteja em andamento. Operações de escrita, como transferências ou atualizações de histórico, requerem exclusividade, bloqueando temporariamente novas leituras até que a modificação seja concluída.

Na implementação do `ServerDatabase`, diferentes tabelas possuem travas dedicadas, (`client_table_lock` protege a tabela de clientes e seus saldos; `transaction_history_lock` controla o histórico de transações; `bank_summary_lock` garante acesso seguro ao sumário global.)

Além da base de dados, o servidor requer sincronização na interface de exibição (`ServerInterface`). Essa classe opera em uma *thread* dedicada que consome mensagens de log e imprime atualizações do estado do banco. Como múltiplas *threads* podem gerar notificações simultaneamente, a fila de mensagens é protegida por um `std::mutex` e uma `std::condition_variable`, garantindo que o método `notifyUpdate()` insira e sinalize as mensagens de forma segura e ordenada.

5 Estruturas e Funções Principais

5.1 Estruturas de Dados

Estrutura de Cliente:

```
struct Client {
    string ip;
    int last_req;
    uint32_t balance;

    Packet last_ack_response;

    Client(const string& client_ip) : ip(client_ip), last_req(0),
        balance(100.0) {}
};
```

Estrutura de Transação:

```
struct Transaction {
    int id;
    string origin_ip;
    int req_id;
    string destination_ip;
    uint32_t amount;

    Transaction(int next_transaction_id, const string& origin_ip,
        int req_id, const string& destination_ip, uint32_t amount)
        : id(next_transaction_id++), origin_ip(origin_ip), req_id
           (req_id), destination_ip(destination_ip), amount(amount
           ) {}
};
```

Estrutura do Banco:

```
struct BankSummary {
    int num_transactions;
    uint32_t total_transferred;
    uint32_t total_balance;
};
```

Essas estruturas foram agrupagadas na classe `ServerDatabase`:

```
class ServerDatabase {
private:
    unordered_map<string, Client> client_table;
    mutable RWLock client_table_lock;

    vector<Transaction> transaction_history;
    mutable RWLock transaction_history_lock;

    BankSummary bank_summary;
    mutable RWLock bank_summary_lock;

    atomic<int> next_transaction_id;
public:
    ...
```

5.2 Principais Funções

- `discoverServer()`: Realiza a descoberta do servidor no lado cliente; envia broadcasts PKT_DISCOVER, aguarda PKT_DISCOVER_ACK com timeout de 1 segundo usando select(), e retorna o endereço IP do servidor; retenta até 50 vezes (MAX_DISCOVERY_ATTEMPTS) em caso de falha. (Subserviço de Descoberta - Cliente)

- `runProcessingLoop()`: Thread dedicada no cliente que processa sequencialmente a fila de comandos; para cada comando, constrói um pacote PKT_REQUEST com seqn incremental, invoca sendRequestWithRetry() e incrementa o contador de sequência apenas após receber ACK válido. (Subserviço de Processamento - Cliente)

- `sendRequestWithRetry()`: Implementa o mecanismo de retransmissão automática (RRA) com timeout configurável; envia requisição ao servidor, aguarda ACK usando select() com timeout de 10ms, valida o seqn do ACK recebido, e retransmite até MAX_RETRIES vezes em caso de perda ou timeout; ao receber ACK válido, encaminha dados para a interface via pushAck(). (Subserviço de Processamento - Cliente)

- `handlePacket()`: Função de delegação que identifica o tipo de pacote recebido (PKT_DISCOVER ou PKT_REQUEST) e encaminha para o handler apropriado; cria threads detached para requisições de processamento, garantindo não-bloqueio do loop principal. (Subserviço de Processamento - Servidor)

- `handleDiscovery()`: Processa pacotes PKT_DISCOVER no servidor; extrai o IP do cliente, registra-o no banco de dados através de addClient(), atualiza o sumário do banco, e envia PKT_DISCOVER_ACK como resposta. (Subserviço de Descoberta - Servidor)

- `handleRequest()`: Função executada em thread dedicada que processa requisições PKT_REQUEST no servidor; valida sequência (seqn), detecta duplicatas e pacotes fora de ordem, executa transações ou consultas de saldo, e envia ACK ao cliente. (Subserviço de Processamento - Servidor)

- `makeTransaction()`: Função central de processamento de transações; valida saldo, atualiza contas de origem e destino, registra no histórico e gera resposta de confirmação. (Subserviço de Processamento - Servidor)

- `inputLoop()`: Thread dedicada no cliente que lê comandos da entrada padrão (stdin); para cada linha válida, extrai o IP destino e o valor da transação usando istringstream, e enfileira o comando através de enqueueCommand(); valida o formato de entrada e continua executando até que running_ seja false ou EOF seja detectado. (Subserviço de Interface - Cliente)

- `outputLoop()`: Thread dedicada no cliente que consome ACKs da fila interna; aguarda notificações via variável de condição, converte endereços IP (servidor e destino) de network order para formato legível usando inet_ntop(), e imprime no formato "timestamp server id_req dest value new_balance".(Subserviço de Interface - Cliente)

- `run()`: Thread principal da interface do servidor; imprime sumário inicial do banco (num_transactions, total_transferred, total_balance), aguarda notificações via variável de condição, consome mensagens da fila interna msgs_, imprime cada linha de log com timestamp, e atualiza o sumário do banco após cada evento processado. (Subserviço de Interface - Servidor)

- `notifyUpdate()`: Adiciona mensagens de log à fila interna de forma thread-safe usando mutex; utilizada pelo processador de requisições (handleRequest) para notificar a interface sobre eventos de transações; dispara a variável de condição para acordar a thread de interface e processar a mensagem imediatamente. (Subserviço de Interface - Servidor)

6 Comunicação Interprocesso

O sistema foi arquitetado sobre o **User Datagram Protocol (UDP)**, utilizando diretamente as primitivas POSIX de comunicação (`socket()`, `bind()`, `sendto()` e `recvfrom()`) para o envio e recepção de datagramas entre processos distintos. Sobre essa base, foi implementado um mecanismo de Requisição–Resposta–ACK (**RRA**), responsável por garantir a ordem e a consistência das transações mesmo em um meio não confiável.

6.1 Fluxo de Operação e Descoberta

O fluxo de comunicação é dividido em duas fases distintas:

- **Descoberta (Broadcast):** O cliente localiza o servidor enviando pacotes `PKT_DISCOVER` no modo broadcast. O servidor, ao receber, registra o novo cliente no banco de dados e responde imediatamente com um pacote `PKT_DISCOVER_ACK` em modo *unicast*.
- **Requisições (Unicast):** Após a descoberta, todas as mensagens subsequentes (transferências e confirmações) utilizam exclusivamente a comunicação ponto-a-ponto (*unicast*).

6.2 Mecanismo de Confiabilidade e Controle de Sequência

A robustez do protocolo é garantida pela combinação de *timeout* no cliente e detecção de duplicidade no servidor.

- **Retransmissão e Bloqueio (Cliente):** O cliente opera de forma bloqueante. Para lidar com perdas, a requisição é monitorada com um *timeout*; se o ACK não for recebido, a requisição é reenviada e o cliente só prossegue com o envio da próxima requisição após a confirmação da anterior.
- **Controle de Duplicidade (Servidor):** O servidor mantém o registro do último número de sequência processado (`last_req`) por cliente. Esta lógica é crucial:
  1. Se o `seqn` recebido for menor ou igual ao `last_req`, o pacote é uma duplicata (retransmissão). O servidor não reprocessa a transação e reenvia o ACK previamente armazenado no *buffer* do cliente.
  2. Se o `seqn` for maior que o `last_req` +**1**, é um pacote fora de ordem (`OOR`) e o servidor responde com o ACK do `last_req`, sinalizando a perda ao cliente para que ele possa se resincronizar.

7 Divisão de Tarefas na Equipe

- **Eduarda Tessari Pereira** Atuou na implementação do subserviço de descoberta, na estruturação do cliente e de seu módulo de requisições, no suporte ao servidor nas rotinas de tratamento de duplicidade, na garantia de atomicidade e consistência, e na construção dos mecanismos de leitura/escrita e uso das primitivas de sincronização.

- **Maximus Borges da Rosa** atuou na definição das estruturas de dados principais e métodos relacionados a elas, e implementação do processamento do lado do servidor. Além disso, estruturou o modelo de controle leitor/escritor e a interface do servidor e trabalhou na refatoração e modularização do projeto.
- **Guilherme D'Ávila Pinheiro** atuou na definição das *estruturas de dados principais* e métodos relacionados a elas, trabalhou na implementação do processamento do lado do servidor, ajustou a interface e estrutura dos pacotes para mostrar todos os dados necessários.
- **Laura Reis Müller** participou do desenvolvimento inicial da estrutura do projeto, revisão de código e testes. Também foi responsável pela redação do relatório final e pela criação dos slides para a apresentação.

8 Dificuldades

1. Gerenciamento de Concorrência e Sincronização de *Threads*

O desafio técnico mais crítico envolveu a correta sincronização das *threads* que gerenciavam as transações Pix. A natureza distribuída e concorrente do processamento de pagamentos levou à ocorrência de dois problemas clássicos de concorrência:

- **Condições de Corrida (*Race Conditions*):** Houve instâncias onde a ordem de execução das *threads* não era determinística, resultando em estados inconsistentes do sistema (por exemplo, saldos incorretos) devido ao acesso simultâneo a recursos compartilhados.
- **Impasse (*Deadlock*):** Ocorreram situações de bloqueio mútuo, notavelmente quando um sub-método tentava adquirir o mesmo ***mutex*** ou **variável de condição** que já estava bloqueada pelo método principal, paralisando o processamento.

Para resolver esses problemas, implementamos um protocolo de sincronização, revisando criticamente o uso de *mutexes* e variáveis de condição para garantir o acesso exclusivo aos recursos críticos e reestruturando a lógica de bloqueio para evitar ciclos de espera circular que levavam a *deadlocks*.

2. Dificuldades na Fase de Testes e Validação

- **Restrições de Ambiente:** A simulação de um ambiente distribuído com mais de uma máquina era limitada aos laboratórios do Instituto. Fora deste ambiente, a validação em múltiplos *hosts* só foi possível em uma máquina individual de um dos membros da equipe, dificultando a replicação e a depuração de problemas específicos de rede.

Para garantir a integridade e a funcionalidade do sistema em um ambiente limitado, foram adotadas duas estratégias principais:

- **Automatização de Testes:** Desenvolvemos *scripts bash* de teste automatizado. Usamo esses *scripts* para verificar rapidamente a integridade do sistema após cada atualização do código-fonte para verificar rapidamente a integridade do sistema.

- **Clientes "Falsos"(*Mock Clients*):** Adicionamos um cliente falso para receber transações, permitindo fazer testes parciais o projeto com somente uma máquina.

3. Coordenação e Modularização do Desenvolvimento

A interligação entre as diversas partes do sistema cliente-servidor e os subserviços exigiu um esforço adicional na divisão de tarefas e na manutenção da coerência do código.

- **Modularização:** O sistema foi projetado com cuidado para manter uma boa modularidade. As implementações dos subserviços (por exemplo, interface, processamento) foram mantidas independentes para minimizar dependências diretas.
- **Técnicas de Colaboração:** Para lidar com as partes mais críticas ou interligadas, utilizamos alocação de tempo de desenvolvimento separado para evitar conflitos de *merge* e praticamos o ***peer-programming*** (programação em pares), o que garantiu a revisão imediata do código e um entendimento compartilhado da lógica complexa do sistema.