# Relatório de Modificações no Código Fonte (T2 Sistemas Distribuídos)

**Heitor Kum, Pedro Cardoso, Pedro Pacheco**

**Objetivo**: Este relatório descreve as modificações realizadas no código fonte original para tornar o sistema distribuído, permitindo que ele seja executado em múltiplos containers Docker. A implementação foi adaptada para utilizar threads e sockets para comunicação entre os processos.

## Modificação 1: Tornar o Código Distribuído

Ambiente Distribuído:

O código foi alterado para suportar um sistema distribuído, onde múltiplas instâncias do ambiente (env) são executadas, ao invés de uma única instância central.
Cada instância do ambiente (env) foi configurada para rodar em *containers* separados no *Docker*. Isso garante que cada parte do sistema funcione de forma independente, mas ainda se comunique entre si de maneira eficiente.

### Execução em Containers Docker:

O código foi adaptado para ser executado em containers *Docker*, com cada ambiente (env) rodando em um *container* isolado.
Isso melhora a escalabilidade e permite que múltiplos componentes do sistema (como lí*deres, aceitores e réplicas*) sejam executados de forma independente, mas ainda se comuniquem via rede, utilizando sockets.
Cada container recebe uma configuração própria e é capaz de se comunicar com outros containers na rede.

## Modificação 2: Classes de Entidades e Processos

Classes para Entidades:

Cada entidade do sistema (como *líder, aceitor e réplicas*) foi convertida em uma classe separada.
Cada classe representa uma entidade distinta, encapsulando o comportamento e as características dessa entidade.

### Herança da Classe Process:

Todas as entidades derivam da classe base Process. A classe Process gerencia a execução de cada instância do processo, garantindo que cada uma delas seja independente e execute sua lógica em paralelo. Com isto, é possível adaptar o comportamento através do método *Handler*.

**Modificação 3: Semáforo**

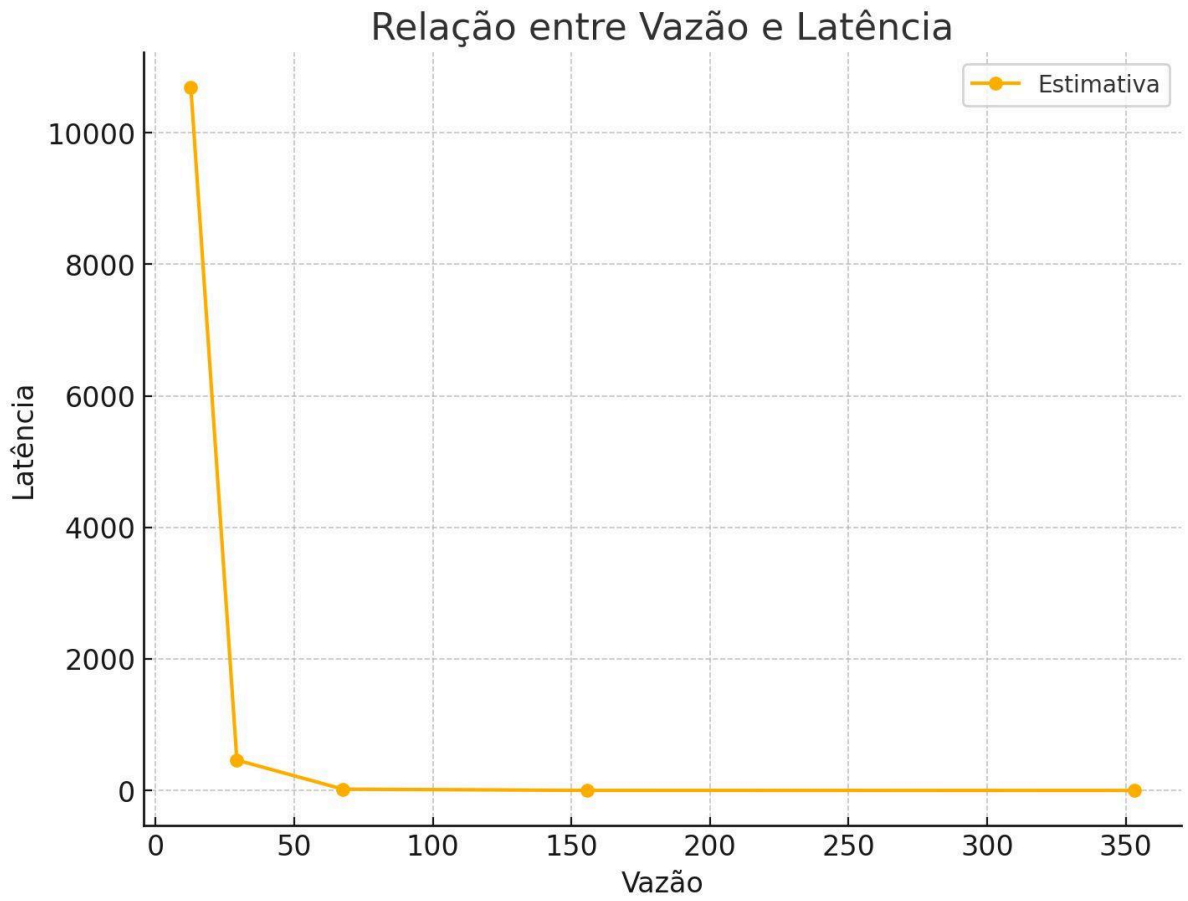
Foi utilizado um semáforo na implementação dos líderes e clientes para manter o envio de mensagens serial.

**Dificuldades na implementação:**

Para implementar este trabalho, utilizamos um código fornecido pelo professor que fazia uso de threads para separar cada entidade(acceptor, leader, commander, etc.). A primeira parte do trabalho foi modificar o código, que fazia o uso de uma execução de `env.py` por execução do algoritmo, para rodar em diferentes máquinas, rodando assim N execução de `env.py`, tendo que mudar muito como o código era estruturado, pois o antigo assumia o conhecimento de todo o sistema por execução, enquanto a implementação do nosso grupo não.
Para chegar nisso, fizemos o uso da função `listen_for_messages` para coordenar todo código. Todas entidades herdaram de `Process` e por isso tinham essa função, que era ativada toda vez que uma mensagem era enviada por `socket`. Então, criamos um handler para cada entidade e, baseado no tipo de nodo que estava consumindo(usamos docker e exportamos as variáveis de ambiente para saber qual container era qual tipo de novo), chamavamos o handler respectivo, que iria rodar o seu algoritmo e enviar(ou não) mensagens de socket para os outros containers.
Outro problema que encontramos foi a utilização do `Docker` para simular múltiplas máquinas, porém o ambiente era um pouco instável e com isso não conseguimos ver logs de nenhuma máquina além do líder.
Foi bem trabalhosa a migração de multithread para diferentes computadores, que não era o enfoque do trabalho, pelo menos assim entendemos nós.

**Resultados:**

**Máquina de teste:**
**Modelo:** MacBook M2 Pro




**Como executar:**
docker-compose up –build (na pasta do projeto)