Universidade Federal de Santa Catarina

INE - Departamento de Informática e Estatística

Programa de Pós-Graduação em Ciência da Computação

INE6116000 - Inteligência Artificial Conexionista		
Aluno:	Victor Rodrigo Leite Magalhães de Almeida	Turma: 2025.1
Atividade:	Protocolo DUR	Data: 10/07/25

1) Introdução

O presente relatório tem por objetivo apresentar a implementação prática do protocolo de replicação *Deferred Update Replication* (DUR) [Pedone e Schiper 2012], conforme especificado trabalho final da disciplina. O protocolo DUR busca assegurar a consistência entre múltiplas réplicas de dados em sistemas distribuídos através de um modelo de adiamento, onde as atualizações são realizadas localmente e propagadas posteriormente para as demais réplicas de forma ordenada. O trabalho visa exercitar conceitos fundamentais da disciplina de Computação Distribuída, como controle de concorrência, comunicação entre processos e replicação de dados, aplicando-os em um cenário que simula um sistema transacional concorrente distribuído.

2) Arquitetura do Sistema

A arquitetura do sistema proposto baseia-se em três tipos de entidades:

- **Cliente**: Responsável por executar transações, que consistem em operações de leitura e escrita. Cada cliente mantém localmente um conjunto de leitura (`rs`) e um conjunto de escrita (`ws`), os quais são enviados ao final da transação para validação.

- **Sequenciador**: Atua como responsável pela difusão atômica. Recebe requisições de `commit` dos clientes, atribui um identificador global (`tx_id`) e difunde a transação ordenadamente para todos os servidores.

- **Servidores**: São réplicas de um banco de dados que armazenam e versionam os dados. Cada servidor realiza o teste de certificação localmente e aplica as operações de escrita caso a transação seja considerada válida.

O sistema comunica-se através de *sockets* TCP, e cada entidade atua de forma concorrente utilizando *threads* independentes. A Figura 1 ilustra o funcionamento das transações dos clientes nos dois tipos de fases existentes. As linhas tracejadas exemplificam as transações de `read` realizada no servidor e as linhas cheias representam as etapas de `commit`. As transações de `write` não estão representadas, pois são realizadas localmente no cliente, antes de ser realizado o `commit`.

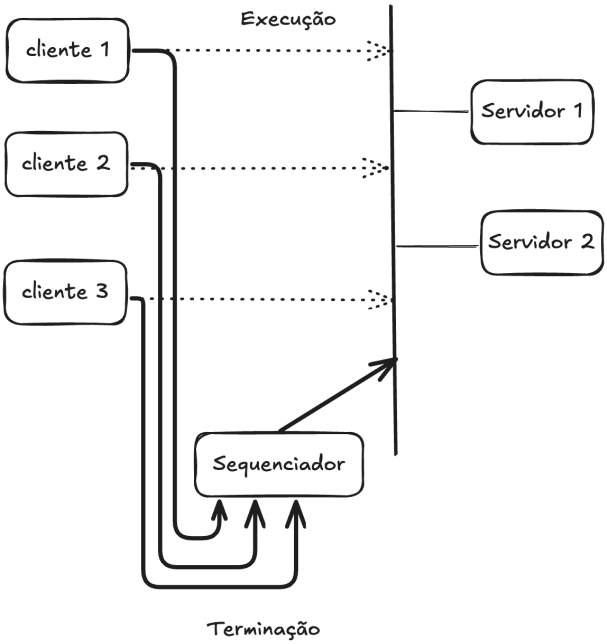

Figura 1 – Comunicação dos componentes do sistema

3) Implementação

A implementação foi realizada em linguagem Python 3, utilizando as bibliotecas padrão `socket` para comunicação em rede e `threading` para execução concorrente. Cada entidade foi modularizada em arquivos distintos:

- `cliente.py` — Contém a implementação da lógica transacional do cliente, incluindo operações de leitura, escrita e commit.
- `sequenciador.py` — Implementa o difusor atômico responsável por atribuir ordem total às transações e disseminá-las para os servidores.
- `servidor.py` — Define a lógica de resposta a leituras, certificação de transações e aplicação condicional das operações no banco de dados local.
- `teste_concorrencia.py` — Script principal que inicializa o sistema e executa os testes concorrentes entre clientes.

O modelo segue os algoritmos de transação e servidor, descritos na literatura de [Mendizabal et.al 2013]. Para o sequenciador, foi assumido que nenhum dos nodos falha. Além da implementação, foram simulados cinco cenários de teste para validação do protocolo.

4) Mapeamento dos Algoritmos e Primitivas para o Código

A seguir, apresenta-se o mapeamento entre os elementos dos algoritmos 3 (Cliente) e 4 (Servidor) conforme definidos no enunciado e suas respectivas implementações no código Python:

4.1) Algoritmo 3 - Cliente

Linha	Descrição no Algoritmo	Implementação em Python
l.2	Seleciona um servidor aleatório	`random.choice(SERVERS)` em `read()`
l.3–4	Executa escrita local no `ws`	`self.ws.append((x, val))` em `write()`
l.7–8	Leitura de item previamente escrito	Leitura é feita diretamente do `ws` se presente, `if item == x` em `read()`
l.10–12	Solicita leitura ao servidor	Envio de mensagem `'type': 'read'` via socket
l.14–15	Solicita commit e difusão ordenada	Envia `tx` via socket ao sequenciador com `tx['type'] = 'commit'`

4.2) Algoritmo 4 - Servidor

Linha / Conceito	Descrição no Algoritmo	Implementação em Python
l.4–5	Espera requisição de leitura	`msg['type'] == 'read'` em `trata_mensagem()`
l.6	Recebe solicitação de commit	`msg['type'] == 'commit'` em `trata_mensagem()`
l.8–12	Executa teste de certificação	Função `certifica(tx)` verifica versões do `rs`
l.15–20	Atualiza banco e incrementa versão	Aplicação de `ws` ao `self.db` com versão incrementada

Também foi realizdo o mapeamento das primitivas descritas no enunciado do trabalho, conforme a seguir nas seções 4.3.

4.3) Primitivas 1:1 e 1:n

As primitivas de comunicação especificadas para este trabalho compreendem interações ponto a ponto (1:1), representadas por `send(m)` e `receive(m)`, e interações de difusão 1:n, representadas por `broadcast(m)` e `deliver(m)`. Na implementação em Python, as primitivas `send(m)` e `receive(m)` são satisfeitas por meio das chamadas `socket.sendall()` e `socket.recv()`,

Tabela 3 – Mapeamento das primitivas de comunicação

Primitiva	Implementação em Python	Satisfeita?
`send(m)`	`socket.sendall()` para cliente-servidor ou sequenciador-servidor	Sim
`receive(m)`	`socket.recv()` no lado do servidor ou sequenciador	Sim
`broadcast(m)`	`for port in server_ports: sendall(tx)` no sequenciador	Sim
`deliver(m)`	`trata_mensagem()` no servidor que processa o commit recebido	Sim

utilizadas tanto nas comunicações entre cliente e servidor para operações de leitura quanto entre o sequenciador e os servidores durante o envio de commits. A primitiva `broadcast(m)` é implementada de forma determinística pelo sequenciador, que transmite a mesma mensagem de commit para todos os servidores replicados, garantindo uma ordem total por meio de um identificador único (`tx_id`). Por fim, a primitiva `deliver(m)` corresponde à recepção e tratamento da transação no lado dos servidores, realizada dentro da função `trata_mensagem()`, assegurando que todos os servidores processem as transações na mesma ordem e cheguem às mesmas decisões de *commit* ou *abort*.

5) Experimentos

Foram definidos quatro casos de teste com o objetivo de validar o comportamento concorrente do protocolo DUR:

1) **Teste 1 — Concorrência simples:** duas transações acessam o mesmo item `x`, com pequenos atrasos para induzir interleaving. Espera-se que apenas uma delas comite.

2) **Teste 2 — Independência de transações:** transações que acessam itens distintos (`a` e `b`). Ambas devem ser comitadas.

3) **Teste 3 — Leitura obsoleta:** uma transação lê o item `z`, outra escreve e comita antes. A primeira deve ser abortada por certificação.

4) **Teste 4 — Três concorrentes em sequência:** três clientes escrevem no mesmo item `m` com atrasos diferentes. Apenas a primeira deve comitar, as demais devem abortar.

5) **Teste 5 — Leitura local após escrita:** o cliente escreve no item `k` e realiza a leitura na mesma transação. A leitura deve ser realizada localmente no `ws`, e não ser realizada uma consulta ao servidor.

Os testes executados demonstraram o comportamento esperado do protocolo DUR.

6) Resultados e Discussão

Os resultados obtidos confirmam que o sequenciador desempenhou corretamente sua função de ordenação global das transações, atribuindo identificadores únicos (`tx_id`) que garantiram

a consistência entre as réplicas. Observou-se ainda que transações cujos conjuntos de leitura estavam desatualizados foram devidamente abortadas pelos servidores durante o teste de certificação, conforme esperado pelo protocolo. Além disso, a utilização da biblioteca `threading` demonstrou-se eficaz na simulação de múltiplos clientes e servidores concorrentes, permitindo a execução paralela e controlada dos cenários de teste.

A arquitetura apresentou robustez frente a cenários de conflito e independência. Destaca-se a importância do controle de versões e do broadcast ordenado como pilares para manter a serialização das transações.

Para trabalhos futuros, a implementação descrita neste documento está disponível e divulgada em formato *open-source*, no *GitHub* do autor deste trabalho.

7) Referências

Pedone, F. e Schiper, N. (2012). *Byzantine fault-tolerant deferred update replication*.

Mendizabal, O. M., Dotti, F. L. (2013). *Model checking the deferred update replication protocol*.

Repositório do projeto: `https://github.com/vrodrigoleite/ine-cd-trabalho-final`