Trabalho Prático 1: Sistema de Impressão Distribuída com Exclusão Mútua usando gRPC, Algoritmo de Ricart-Agrawala e Relógios Lógicos de Lamport

Camila M. Lopes[1], **Dã P. Gonçalves**[1], **Hanna R. Chaves**[1], **Laura C. Costa**[1] **Lucas S. G. Castro**[1]

[1]Instituto de Ciências Exatas e Informática
Pontifícia Universidade Católica e Informática (PUC-MG)

lopes.camilamoreira@gmail.com

dapetronilho@gmail.com,

hannarchaves@gmail.com

Laura.costa.1204819@sga.pucminas.br

lsgcastro@sga.pucminas.br

Abstract. *The Distributed Printing project aims to implement, in a practical and didactic way, the Ricart–Agrawala mutual exclusion algorithm, using Lamport logical clocks to coordinate access between multiple clients to the same shared resource — in this case, a simulated printer. The system was developed with Python and gRPC, allowing communication between distributed processes in an efficient and organized manner.*

Resumo. *O projeto de Impressão Distribuída tem como objetivo implementar, de forma prática e didática, o algoritmo de exclusão mútua de Ricart–Agrawala, utilizando relógios lógicos de Lamport para coordenar o acesso entre múltiplos clientes a um mesmo recurso compartilhado — no caso, uma impressora simulada. O sistema foi desenvolvido com Python e gRPC, permitindo a comunicação entre processos distribuídos de maneira eficiente e organizada.*

1. Explicação da arquitetura e funcionamento do código

O sistema é composto por dois tipos principais de processos: um servidor de impressão e vários clientes distribuídos que disputam o acesso à impressora.

O servidor de impressão é simples (“burro”), sendo responsável apenas por receber solicitações e simular a impressão. Ele está implementado na classe PrintingServiceServicer, no arquivo (printer_server.py), e expõe o serviço via gRPC através da função serve. Quando recebe uma requisição SendToPrinter, ele aguarda cerca de dois segundos (simulando o tempo de impressão) e retorna uma resposta de sucesso ao cliente.

Os clientes são os responsáveis pelo controle de exclusão mútua e estão implementados em (printing_client.py). Cada cliente atua simultaneamente como solicitante de impressão e como pequeno servidor local, capaz de receber requisições de outros clientes. Esse servidor local é implementado pela classe MutualExclusionServicer, que disponibiliza os métodos RequestAccess e ReleaseAccess via gRPC.

Quando um cliente deseja imprimir, ele primeiro atualiza seu relógio de Lamport, muda o estado interno para WANTED, e envia requisições de acesso (AccessRequest) a todos os outros clientes. Cada mensagem enviada inclui o timestamp lógico e o identificador do cliente, para que os outros possam decidir quem tem prioridade no uso da impressora.

Os clientes que recebem a solicitação comparam os valores (timestamp, client_id) e decidem se concedem ou adiam o acesso. O critério de prioridade segue a ordem lexicográfica: o cliente com o menor timestamp (e, em caso de empate, menor ID) ganha prioridade.

Uma parte importante da arquitetura é o uso de threading.Condition, uma variável de condição compartilhada que permite a sincronização entre threads locais. Caso um cliente já esteja utilizando a impressora (estado HELD), as chamadas gRPC de entrada (RequestAccess) ficam bloqueadas até que o cliente libere a seção crítica. Quando a impressão termina, o cliente muda o estado para RELEASED e chama notify_all(), liberando os outros processos que estavam aguardando. Assim, o próximo cliente que estava esperando pode prosseguir e realizar sua própria impressão.

2. Análise do algoritmo de Ricart-Agrawala implementado

O algoritmo de Ricart–Agrawala (RA) foi originalmente proposta por Ricart e Agrawla (1981) no artigo "*An Optimal Algorithm for Mutual Exclusion in Computer Netwaorks*"[Ricart and Agrawala 1981]. Esse algoritmo é um protocolo de exclusão mútua distribuída do tipo *permission-based*, utilizado para coordenar o acesso de processos distribuídos a um recurso compartilhado, isto é, uma seção crítica, sem a necessidade de um coordenador central.

Dessa forma, é possível afirmar que seu princípio baseia-se na troca de mensagens entre os processos e na utilização de relógios lógicos de Lamport para a ordenação dos eventos.

2.1. Visão geral teórica

Quando um processo deseja acessar a seção crítica, ocorre a seguinte sequência de estados:

1. **Incrementa** seu relógio lógico e envia uma mensagem **`REQUEST`** para todos os outros processos.
2. Cada processo que **recebe** um pedido responde com **`REPLY`** se não estiver na seção crítica ou se possuir *timestamp* maior que o do solicitante.
3. Caso contrário, o processo **adia** a resposta até sair da seção crítica.
4. O solicitante entra na seção crítica apenas após **receber** respostas de todos os demais.

A complexidade de mensagens do algoritmo é $2 \times (N - 1)$ por acesso à seção crítica. E o algoritmo RA garante a exclusão mútua e ausência de *deadlock*, desde que todos os processos estejam ativos e a comunicação seja confiável.

2.2. Arquitetura da implementação

A implementação foi desenvolvida em Python utilizando gRPC para comunicação entre processos distribuídos. E é composta pelos seguintes módulos:

Table 1. Arquivos e funções na implementação.

Arquivo	Função
`print_pb2.py` `print_pb2_grpc.py`	Definem as mensagens e serviços *gRPC* utilizados na comunicação entre os processos distribuídos.
`printing_client.py`	Implementa o algoritmo de Ricart–Agrawala, responsável pela coordenação entre os clientes e controle de acesso ao recurso crítico.
`printer_server.py`	Simula o servidor de impressão (recurso crítico), recebendo e processando as solicitações enviadas pelos clientes.

Assim, os cliententes foram divididos de tal forma que representassem um processo que alterna entre três estados:

- `RELEASED` - não deseja entrar na seção crítica;
- `WANTED` - deseja entrar e aguarda permissão;
- `HELD` - está na seção crítica.

3. Resultados dos testes realizados

3.1. Testes Realizados

Foram realizados dois testes experimentais para avaliar o funcionamento do algoritmo de RA implementado. As execuções envolveram três clientes (Cliente 1, Cliente 2 e Cliente 3) e um servidor central de impressão, que representa a seção crítica. Foram observados dois cenários principais como observamos na Figura 1: cenário 1 com requisições simultâneas (concorrência) e cenário 2 com execução sequencial (sem concorrência).

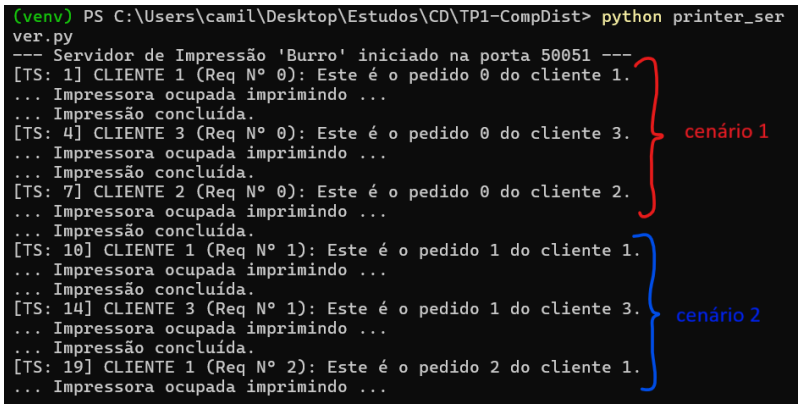

Figure 1. Log do Servidor

3.1.1. Cenário 1 - Concorrência entre processos

Neste cenário, o Cliente 3 iniciou a impressão primeiro, enquanto os Clientes 1 e 2 solicitaram acesso durante sua execução. O algoritmo coordenou corretamente os pedidos simultâneos, deferindo e concedendo acessos conforme a ordem dos timestamps lógicos.

Os logs mostram que, enquanto o Cliente 3 estava na seção crítica, o Cliente 2 teve seu pedido adiado. Dessa forma, após o Cliente 3 liberar o recurso, o algoritmo avaliou os timestamps e concedeu prioridade ao Cliente 1, que possuía menor valor lógico. O Cliente 1, então, entrou na seção crítica, realizou a impressão e liberou o acesso para o Cliente 2. Esse comportamento confirma a ordenação justa das requisições e a preservação da exclusão mútua mesmo sob concorrência.

Em sistemas distribuídos, a concorrência ocorre quando vários processos tentam acessar simultaneamente um recurso sem uma noção global de tempo. Para lidar com isso, o algoritmo de Ricart–Agrawala utiliza os relógios lógicos propostos por Lamport [Lamport 1978], que permitem ordenar eventos de forma consistente entre processos distintos. Assim, mesmo sem um relógio físico comum, o sistema garante que o processo com o menor timestamp lógico tenha prioridade de acesso, mantendo justiça e sincronização na execução.

As figuras de log (Figura 2 e Figura 3) ilustram visualmente a sequência de eventos e demonstram a correta coordenação entre os clientes e o servidor.

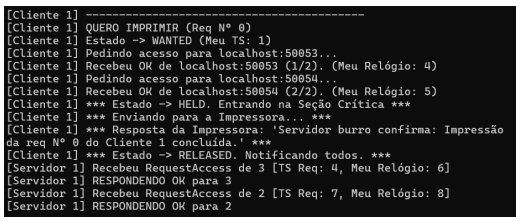

Figure 2. Cliente 1

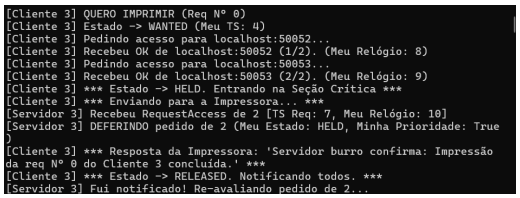

Figure 3. Cliente 3

Figure 4. Comportamento dos clientes no Cenário 1

3.1.2. Cenário 2 - Funcionamento básico sem concorrência

No segundo cenário, apenas um cliente por vez solicitou acesso à impressora. O objetivo foi verificar o funcionamento básico do algoritmo na ausência de disputas simultâneas.

Como observamos na figura o Cliente 1 enviou o pedido, recebeu as permissões dos demais clientes e entrou na seção crítica. O servidor burro processou as impressões sequencialmente, confirmando a correta operação do sistema em situações de baixa concorrência. Nenhum caso de espera indefinida ou conflito foi observado.

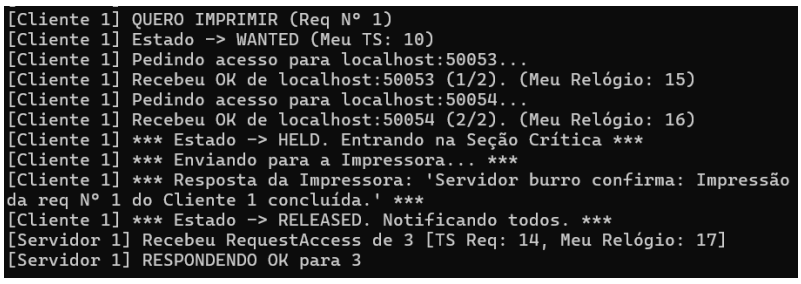

Figure 5. Cliente 1 no cenário 2

3.2. Propriedades observadas

Durante os testes realizados, observou-se que a implementação do algoritmo de RA atendeu às principais propriedades esperadas de um mecanismo de exclusão mútua distribuída. A Tabela 2 a seguir apresenta as propriedades verificadas, acompanhadas das respectivas observações e resultados.

Table 2. Propriedades verificadas na implementação

Propriedade	Observação	Resultado
Exclusão mútua	Apenas um cliente por vez na seção crítica	Satisfeita
Deadlock	Não observado em execuções normais	Satisfeita
Justiça (ordenamento)	Baseada em timestamps lógicos	Satisfeita
Tolerância a falhas	Clientes inativos bloqueiam o sistema	Não implementada

A propriedade de exclusão mútua foi plenamente observada: em todas as execuções, apenas um cliente por vez teve acesso ao recurso crítico, garantindo a integridade do sistema de impressão. Esse comportamento decorre diretamente do protocolo de espera por respostas de todos os outros processos antes da entrada na seção crítica.

No que se refere à ausência de deadlocks, não foram detectados bloqueios permanentes durante as execuções. Isso ocorre porque o uso correto dos relógios lógicos de Lamport assegura uma ordenação total das requisições, evitando dependências circulares entre os processos.

A propriedade de justiça (ou *fairness*) também foi mantida, uma vez que as requisições são tratadas na ordem em que ocorrem, de acordo com seus *timestamps* lógicos. Dessa forma, processos que solicitam acesso mais cedo são atendidos primeiro, evitando o fenômeno de starvation (fome ou espera indefinida).

Por outro lado, a tolerância a falhas não foi implementada na versão atual. Caso um cliente falhe ou fique inativo, os demais processos permanecem aguardando respostas indefinidamente, impedindo o avanço do sistema. Essa limitação é comum em implementações básicas do algoritmo e pode ser mitigada com a inclusão de mecanismos de tempo limite (timeout) e detecção de falhas.

3.3. Conclusão e sugestões

Os testes realizados permitiram avaliar, de forma prática, o comportamento do algoritmo de Ricart–Agrawala em diferentes condições de execução. Tanto no cenário sem concorrência quanto no cenário com múltiplos processos competindo pelo mesmo recurso (impressão), a implementação demonstrou ser capaz de garantir a exclusão mútua, assegurando que apenas um cliente acesse a seção crítica por vez. A ordenação dos eventos foi conduzida corretamente pelos *timestamps* lógicos, mantendo a justiça e a consistência entre as requisições, mesmo em situações de simultaneidade.

Além disso, não foram observados casos de *deadlock* ou inconsistências na liberação de acessos, o que comprova a correção funcional do algoritmo implementado.

Os resultados confirmam, portanto, que o uso de mensagens do tipo `REQUEST` e `REPLY`, aliado ao controle de estados e relógios lógicos, foi suficiente para coordenar o acesso distribuído de forma ordenada e previsível.

Todavia, apesar de o algoritmo ter funcionado corretamente e cumprido seus objetivos principais, a análise revelou algumas limitações que afetam sua escalabilidade e robustez em ambientes distribuídos mais complexos.

- Falta de tratamento de falhas e *timeouts*.
- Ausência de fila explícita de pedidos adiados.
- Chamadas RPC síncronas que aumentam a latência.
- Escalabilidade limitada devido ao bloqueio de threads.

Como melhorias futuras, recomenda-se a implementação de *timeouts* para detecção de falhas e liberação automática de recursos, bem como o uso de comunicação assíncrona (com futures ou async gRPC), permitindo que a troca de mensagens ocorra em paralelo à execução local. Além disso, a criação de uma fila de requisições deferidas tornaria o comportamento mais próximo da formulação original do algoritmo de Ricart–Agrawala, aumentando a clareza, a escalabilidade e a tolerância a falhas da aplicação.

4. Dificuldades encontradas e soluções adotadas

4.1. Servidor "Burro" – Exclusão Mútua sem Coordenação Central

O enunciado estabeleceu que o servidor de impressão deveria ser "burro", ou seja, não poderia participar do processo de exclusão mútua. Essa restrição contrariava o modelo tradicional cliente-servidor, no qual o servidor atua como coordenador central do acesso à seção crítica. Dessa forma, tornou-se necessário desenvolver uma abordagem totalmente descentralizada, em que os próprios clientes realizassem a coordenação entre si.

Portanto, como forma de resolver essa limitação, o servidor foi configurado apenas para oferecer o serviço de impressão (PrintingService), sem qualquer lógica de controle de acesso. Cada cliente passou a desempenhar simultaneamente os papéis de servidor e cliente gRPC, implementando o MutualExclusionService para coordenar diretamente entre si. Assim, foi estabelecida uma arquitetura peer-to-peer, na qual o servidor atua apenas como um recurso compartilhado e passivo.

4.2. Implementação do "Defer" no Algoritmo de Ricart-Agrawala

Durante o desenvolvimento, verificou-se que a implementação do algoritmo de Ricart-Agrawala exigia que um processo adiasse (deferisse) respostas em determinadas situações, como quando já estava na seção crítica (HELD) ou quando desejava acessá-la (WANTED) e possuía prioridade sobre os demais. Essa necessidade trouxe complexidade à implementação em Python, especialmente pela possibilidade de ocorrerem deadlocks ou race conditions devido à concorrência entre threads.

Para lidar com esse desafio, foi empregada a estrutura Condition Variables (threading.Condition), que possibilita o controle da sincronização entre as threads. Também foi implementado um loop de espera que reavalia continuamente o estado do processo após cada notificação recebida. Além disso, a prioridade entre processos foi definida por meio de uma tupla (timestamp, client_id), garantindo desempates determinísticos e a correta ordenação dos acessos à seção crítica.

4.3. Sincronização dos Relógios Lógicos de Lamport

Outro ponto crítico do projeto foi manter a sincronização dos relógios lógicos entre os processos distribuídos, conforme as três regras propostas por Lamport: incrementar o relógio antes de cada evento local, anexar o timestamp a todas as mensagens enviadas e atualizar o relógio ao receber mensagens de outros processos. O desafio consistiu em garantir que todas as etapas do sistema respeitassem essas regras de forma consistente, evitando discrepâncias temporais entre os clientes.

Dessa forma, a solução adotada consistiu em incrementar o relógio lógico antes de cada mudança de estado para WANTED e antes do envio de qualquer mensagem. Todas as mensagens trocadas entre os clientes — AccessRequest, AccessResponse e ReleaseAccess — passaram a incluir o timestamp associado, e, ao receber qualquer mensagem, o cliente atualiza seu relógio utilizando a regra clock = max(local, recebido) + 1. Com isso, assegurou-se a coerência temporal e a correta ordenação dos eventos no ambiente distribuído.

4.4. Comunicação Bidirecional – Cliente atuando como Servidor e Cliente

A necessidade de cada cliente atuar simultaneamente como servidor e cliente gRPC também representou um grande desafio técnico. Isso porque o Python, por padrão, adota um modelo single-threaded, o que dificulta a execução de múltiplas funções de forma paralela em uma única instância. Era necessário, portanto, encontrar uma maneira de permitir que cada cliente recebesse e enviasse requisições ao mesmo tempo.

Para solucionar essa questão, foram utilizadas threads independentes, permitindo a execução paralela das diferentes funções. Uma thread foi dedicada à execução do servidor gRPC de cada cliente, enquanto a thread principal ficou responsável pelo envio das requisições de impressão. Adicionalmente, uma terceira thread foi criada para exibir em tempo real o status das operações. No servidor, implementou-se o ThreadPoolExecutor, possibilitando o tratamento simultâneo de múltiplas requisições e garantindo melhor desempenho e responsividade do sistema.

4.5. Envio de ReleaseAccess e Prevenção de Deadlocks

Durante a implementação do algoritmo de Ricart-Agrawala, identificou-se que, após sair da seção crítica, o processo deveria liberar o acesso para os demais. A versão inicial, entretanto, notificava apenas as threads locais, o que poderia causar deadlocks, pois os outros clientes permaneceriam aguardando indefinidamente a liberação do recurso.

Para resolver esse problema, foi adicionada a comunicação explícita entre os clientes após a liberação da seção crítica. Assim, cada processo passou a enviar mensagens ReleaseAccess via RPC para todos os demais clientes, além de notificar as threads locais por meio do método cv.notify_all(). Antes de cada envio, o relógio lógico de Lamport é incrementado, garantindo a consistência temporal entre as mensagens. Essa abordagem assegura que todos os clientes atualizem corretamente seus relógios e evita bloqueios mútuos no sistema.

References

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565.

Ricart, G. and Agrawala, A. K. (1981). An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17.