Relatório Técnico: Sistema de Impressão Distribuída

Vitor L. Oliveira, Rafael Fleury, Mourão

[1]Instituto de Ciências Exatas e Informática (ICEI) – PUC Minas
Departamento de Ciência da Computação – Belo Horizonte, Brasil

rafael.barcellos@sga.pucminas.br, vitor.oliveira.14544667@sga.pucminas.br

Abstract. *Este relatório técnico apresenta o desenvolvimento de um Sistema de Impressão Distribuída baseado em comunicação gRPC e implementado em Python 3. O sistema foi projetado com uma arquitetura composta por um servidor de impressão passivo e múltiplos clientes inteligentes, responsáveis pela coordenação de acesso ao recurso compartilhado de impressão. A exclusão mútua é garantida por meio do algoritmo de Ricart-Agrawala [1] aliado aos relógios lógicos de Lamport [2], assegurando a ordem causal dos eventos e a correção da execução concorrente. O trabalho inclui análise da arquitetura, protocolo de comunicação, mecanismos de sincronização e resultados experimentais de testes de concorrência e falha, demonstrando a robustez e a consistência do sistema.*

1. Introdução

O sistema de impressão distribuída proposto implementa um ambiente no qual múltiplos clientes coordenam o acesso a um recurso compartilhado (a impressora) sem a necessidade de um servidor central de controle. O objetivo é garantir exclusão mútua e ordenação lógica de eventos, assegurando que apenas um cliente por vez utilize o serviço de impressão.

Para isso, foi adotada uma arquitetura do tipo *servidor burro e clientes inteligentes*, em que o servidor apenas executa as requisições de impressão recebidas, enquanto os clientes implementam os algoritmos de coordenação e sincronização.

2. Arquitetura do Sistema

A arquitetura é composta por dois componentes principais: o **Servidor de Impressão** e os **Clientes Inteligentes**. O servidor, implementado em `printer_server.py`, é um processo simples que recebe mensagens via gRPC, imprime o conteúdo recebido e simula um atraso de 2–3 segundos. Ele não realiza nenhum controle de concorrência.

Os clientes, definidos em `printing_client.py`, são responsáveis pela coordenação do acesso ao servidor. Cada cliente atua tanto como um servidor gRPC, capaz de receber requisições de outros clientes, quanto como cliente gRPC, podendo se comunicar com o servidor de impressão e com seus pares. Para coordenar o acesso ao recurso, os clientes utilizam o **Algoritmo de Ricart-Agrawala** [1] e os **Relógios de Lamport** [2], garantindo exclusão mútua e ordenação dos eventos.

3. Protocolo de Comunicação (gRPC)

A comunicação entre processos é realizada via **gRPC** [3], conforme definido no arquivo `proto/printing.proto`. Dois serviços principais foram especificados:

PrintingService Implementado pelo servidor de impressão, responsável por receber e processar mensagens via RPC `SendToPrinter`.

MutualExclusionService Implementado pelos clientes, oferecendo os RPCs `RequestAccess` e `ReleaseAccess`, utilizados para coordenar o acesso à seção crítica.

Exemplo de definição dos RPCs:

```
rpc SendToPrinter (PrintRequest) returns (PrintResponse);
rpc RequestAccess (AccessRequest) returns (AccessResponse);
rpc ReleaseAccess (AccessRelease) returns (Empty);
```

4. Algoritmo de Ricart-Agrawala

O algoritmo de Ricart-Agrawala [1] foi utilizado para gerenciar a entrada e saída da seção crítica, permitindo que apenas um cliente acesse o recurso compartilhado por vez. Esse método distribui a responsabilidade de coordenação entre os próprios processos, eliminando a necessidade de um servidor central, e garante que cada solicitação de acesso seja tratada de maneira ordenada e justa.

Cada cliente pode estar em um dos seguintes estados:

- **RELEASED**: não deseja acessar o recurso;
- **WANTED**: deseja acessar o recurso e aguarda permissão dos demais;
- **HELD**: obteve todas as permissões e está na seção crítica.

O processo de requisição de acesso inicia-se com a mudança do estado para `WANTED` e incremento do relógio de Lamport. O cliente então envia mensagens `AccessRequest` a todos os demais e aguarda as respostas. Ao receber todas, o estado é alterado para `HELD`, e o cliente executa a operação de impressão. Após a conclusão, o cliente libera o recurso com `release_access()`, alterando seu estado para `RELEASED` e respondendo as requisições que estavam adiadas.

5. Relógios Lógicos de Lamport

A ordenação causal das requisições é garantida pela classe `LamportClock`, baseada no modelo proposto por Lamport [2]. O método `increment()` é invocado antes de eventos locais, e o método `update()` é acionado ao receber mensagens externas, conforme a regra:

$$T = \max(T_{local}, T_{remoto}) + 1$$

Além disso, a função `compare_timestamps()` é utilizada para desempatar requisições simultâneas, priorizando o cliente com menor ID.

6. Execução e Testes

Os testes foram conduzidos em múltiplos terminais para validar os diferentes cenários do sistema. Primeiramente, o código gRPC é gerado com:

```
bash scripts/generate_proto.sh
```

Em seguida, o servidor é iniciado:

```
./scripts/run_server.sh
```

E finalmente os clientes:

```
./scripts/run_client.sh 1
./scripts/run_client.sh 2
./scripts/run_client.sh 3
```

Os testes contemplaram cenários com um único cliente e com múltiplos clientes. Em todos os casos, o sistema garantiu a exclusão mútua e retomou o funcionamento após falhas, demonstrando resiliência e robustez.

7. Resultados dos Testes

Para avaliar o comportamento do sistema em diferentes cenários de concorrência, foram realizados dois experimentos principais:

7.1. Cenário 1: Um Cliente Conectado

Neste teste, apenas um cliente foi executado. O resultado mostra que o acesso ao servidor de impressão ocorre de forma direta, sem necessidade de coordenação ou espera por outros processos, já que não há concorrência. A Figura 1 apresenta o log completo da execução.

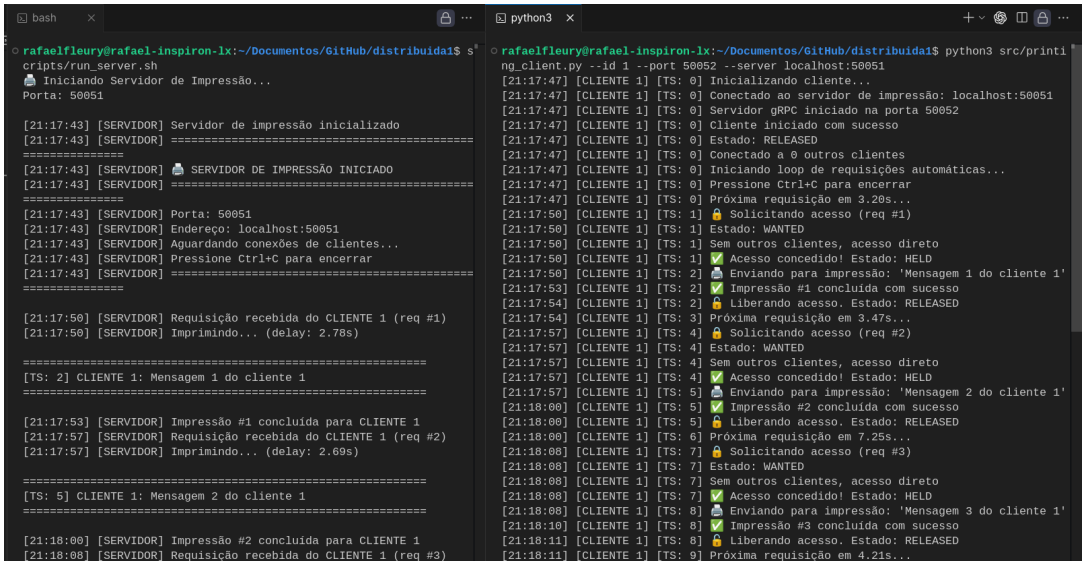

Figure 1. Execução com apenas um cliente. O cliente acessa o servidor diretamente, realizando múltiplas impressões sequenciais.

7.2. Cenário 2: Três Clientes Concorrendo pelo Recurso

No segundo cenário, três clientes foram executados simultaneamente, compartilhando o mesmo servidor de impressão. Como se observa na Figura 2, o algoritmo de Ricart-Agrawala garantiu a exclusão mútua — apenas um cliente por vez entrou na seção crítica. As mensagens de adiamento, liberação e requisição ilustram a troca ordenada de mensagens entre os clientes e a manutenção da consistência dos relógios lógicos.

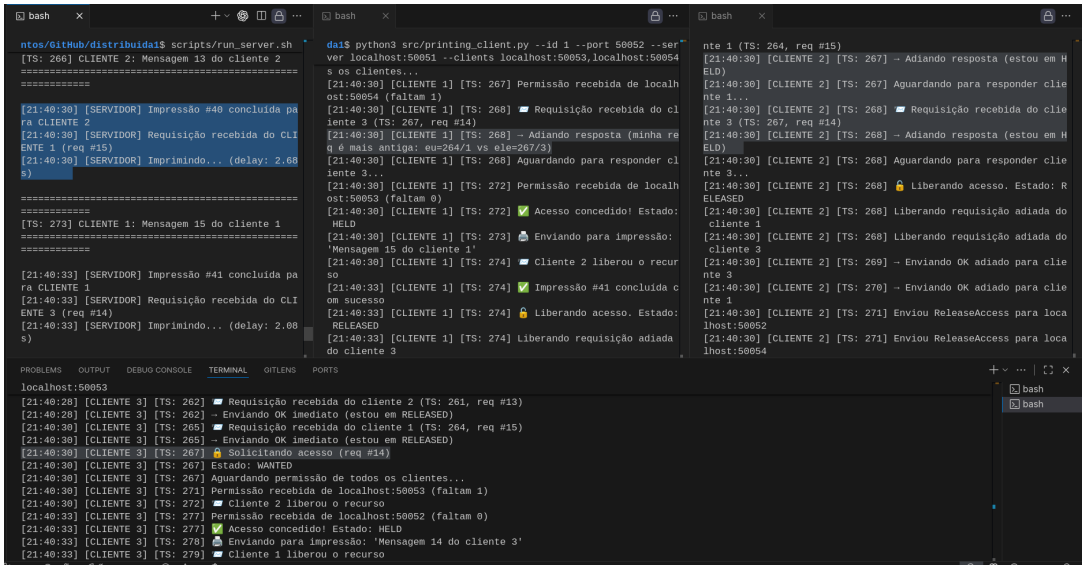

Figure 2. Execução com três clientes. O algoritmo distribui o acesso ao recurso de forma coordenada e livre de conflitos.

7.3. Análise

Os resultados comprovam o correto funcionamento do sistema nos dois cenários:

- Em ambiente sem concorrência, o acesso é imediato e contínuo;
- Em ambiente distribuído, as requisições são ordenadas de acordo com os relógios de Lamport e processadas sem sobreposição;
- Nenhum caso de deadlock ou violação da exclusão mútua foi observado;
- O sistema apresentou comportamento estável mesmo sob múltiplas conexões simultâneas.

8. Desafios e Soluções

Durante o desenvolvimento, foram enfrentadas diversas dificuldades técnicas:

Cliente único bloqueado: O cliente ficava preso aguardando respostas inexistentes. A solução foi adicionar uma verificação em `request_access()` que libera imediatamente o evento quando não há outros clientes.

Deadlock por falha de cliente: Em caso de falha, o sistema aguardava indefinidamente uma resposta. Foi adicionada uma captura de exceções gRPC para remover clientes inativos da lista de espera.

Adiamento de respostas: O Ricart-Agrawala exige que certas respostas sejam adiadas. Para isso, utilizou-se `threading.Event`, bloqueando apenas a thread correspondente até a liberação do recurso.

Condições de corrida: O acesso simultâneo a variáveis compartilhadas foi controlado com `threading.Lock`, garantindo atomicidade e consistência.

9. Conclusão

O sistema de impressão distribuída desenvolvido atendeu plenamente aos objetivos propostos. A arquitetura descentralizada e o uso combinado dos algoritmos de Ricart-Agrawala e Lamport garantiram um controle eficaz de exclusão mútua e ordenação causal, sem dependência de um coordenador central. Os resultados experimentais comprovaram o funcionamento correto e resiliente da solução, mesmo em cenários de concorrência intensa ou falha de componentes.

References

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

[2] Lamport, Leslie. *Time, clocks, and the ordering of events in a distributed system.* In: **Concurrency: the Works of Leslie Lamport**, pp. 179–196, ACM, 2019.

[3] The gRPC Authors. *gRPC Documentation*. Disponível em: https://grpc.io/. Acesso em: 28 out. 2025.