

# Implementação de Exclusão Distribuída

Clara Ribeiro Barreto, Kaio da Silva dos Santos, João Pedro Covello Valente

*Centro Federal de Educação Tecnológica Celso Suckow da Fonseca - CEFET/RJ*

## 1. Introdução

Este trabalho tem como objetivo a implementação e avaliação do algoritmo centralizado de exclusão mútua em sistemas distribuídos. Nesse modelo, um processo especial, chamado Coordenador, é responsável por receber as requisições dos processos interessados em acessar a região crítica e conceder ou negar esse acesso conforme a ordem de chegada, mantendo uma fila interna.

A proposta deste projeto inclui a construção do sistema completo com comunicação via sockets e a criação de um ambiente de testes com múltiplos processos que solicitam acesso concorrente. O programa tem 3 tipos de mensagem -> **Request**: mensagem enviada por um processo para solicitar acesso a região crítica; **Grant**: mensagem enviada pelo coordenador dando acesso a região crítica; **Release**: mensagem enviada por um processo ao sair da região crítica.

Houve a implementação de um coordenador *multi-threaded* responsável por gerenciar a fila de requisições, emitir o Grant e registrar eventos. Nessa implementação, vários processos clientes solicitam repetidamente acesso a região crítica e escrevem em um arquivo compartilhado.

## 2. Decisões de projeto

O projeto foi dividido em dois papéis principais: coordenador e processo. O arquivo “invocador.py” implementa ambos, diferenciando o comportamento via argumento de linha de comando (--role) durante sua inicialização. Para automatizar os experimentos um *script* de rotina foi criado para garantir que o coordenador esteja em execução para então iniciar todos os processos sequencialmente. As entradas padronizadas ajudam a manter o controle da quantidade de requisições que serão realizadas por cada processo e a rotina ajuda a criar identificadores únicos para cada processo além de garantir que cada processo tenha a sua própria execução isolada com seu próprio conjunto de variáveis e espaço de memória.

Para fim de criação e controle dos processos as bibliotecas *threading* e *subprocess* foram utilizadas por proverem, respectivamente, um mecanismo de exclusão mútua para acesso de recursos, e criação de processos filhos independentes uns dos outros. O padrão de comunicação empregado foi o socket TCP devido à sua versatilidade e facilidade de implementação, além de permitir que quaisquer processos possam ser adicionados a qualquer momento para conectar ao socket e se comunicar com o coordenador de processos para acessar o recurso compartilhado.

Todas as mensagens trocadas têm tamanho fixo de 10 *bytes*. O formato é:

- Início: identificador do tipo de mensagem (1 para REQUEST, 2 para GRANT, 3 para RELEASE)
- Identificador do processo
- Conteúdo da mensagem, simulado por uma sequência aleatória de dígitos para completar o tamanho

Exemplo: 1 | 5 | 1234567

O funcionamento do processo pode ser descrito basicamente da seguinte forma:

- Cada processo conecta-se ao coordenador e, em loop, solicita acesso à região crítica (REQUEST), aguarda permissão (GRANT), executa a região crítica (escreve no “resultado.txt” com *timestamp* e id), espera 2 segundos, e libera a região crítica (RELEASE). Repete esse ciclo 5 vezes.
- O identificador do processo é passado como argumento e é único para cada instância.

Enquanto o funcionamento do coordenador:

- **Multi-threaded:** Utiliza threads, que no contexto de *python* são executadas apenas 1 por vez pelo processo mostrando sua transparência para apresentar como um funcionamento único e simultâneo as funções de aceitar e gerenciar as requisições, e gerenciar a interface de comandos.
- **Fila de Pedidos:** Utiliza uma fila protegida por *threading.Lock* para garantir sincronização entre threads.
- **Controle de GRANT:** Apenas um processo pode estar na região crítica por vez. O coordenador só envia novo GRANT após receber RELEASE do processo atual.
- **Log:** Todas as mensagens recebidas e enviadas são registradas em “coordenador\_log.txt” com *timestamp*, tipo, id do processo e conteúdo.
- **Interface:** Permite consultar a fila de pedidos, contagem de atendimentos por processo e encerrar a execução, diretamente por comandos do terminal.

No seguinte trecho de código é apresentada a lógica empregada para o controle do coordenador ao receber uma mensagem de algum processo durante o momento em que algum processo esteja executando a região crítica:

```
if msg_type == '1': # REQUEST
    with self.lock:
        log_message(self.logfile, msg_type, process_id, f"RECEBIDO: {msg}")
        self.request_queue.put((process_id, conn))
        if self.request_queue.qsize() == 1:
            self.enviar_grant(process_id, conn)
elif msg_type == '3': # RELEASE
    with self.lock:
        if not self.request_queue.empty():
            self.request_queue.get()
        log_message(self.logfile, msg_type, process_id, f"RECEBIDO: {msg}")
```

```

self.attended_count[process_id] = self.attended_count.get(process_id, 0) + 1
if not self.request_queue.empty():
    prox_id, prox_conn = self.request_queue.queue[0]
    self.enviar_grant(prox_id, prox_conn)

```

### 3. Avaliação dos Resultados

Após a execução da aplicação, temos acesso ao “coordenador\_log.txt”, que detalha um registro de todas as mensagens trocadas entre o coordenador e os processos, incluindo os metadados já mencionados no tópico anterior. Da mesma forma, temos acesso ao “resultado.txt”, um registro cronológico de quando cada processo entrou na região crítica.

A análise dos arquivos de log e resultado demonstra que o sistema implementado atende plenamente aos requisitos de exclusão mútua distribuída. Primeiramente, é possível observar no trecho abaixo do arquivo “resultado.txt” que os processos 76, 13, 54, 23 e 16 acessam a região crítica em ciclos regulares e intercalados, sem sobreposição de *timestamps*, o que comprova a exclusão mútua adequada por parte do coordenador.

Trecho do “resultado.txt” exemplificando seis ciclos completos:

```

76 2025-07-17 01:28:30.120577
13 2025-07-17 01:28:31.827072
54 2025-07-17 01:28:34.316815
23 2025-07-17 01:28:36.936859
16 2025-07-17 01:28:38.678831
76 2025-07-17 01:28:41.030645

```

Em “resultado.txt”, também é perceptível que o algoritmo implementou 3 repetições por processo como orientado, não favorecendo nenhum processo em detrimento de outros.

No “coordenador\_log.txt”, cada *request* (mensagem do tipo 1) recebida é registrada precisamente, seguida pelo envio do *grant* (mensagem do tipo 2) e depois o *release* (mensagem do tipo 3).

Trecho de “coordenador\_log.txt” exemplificando:

```

2025-07-17 01:28:30.119990 | 1 | 76 | RECEBIDO: 1|76|89820
2025-07-17 01:28:30.120578 | 2 | 76 | ENVIADO: 2|76|45299
2025-07-17 01:28:30.124454 | 1 | 13 | RECEBIDO: 1|13|90201
2025-07-17 01:28:30.125977 | 1 | 54 | RECEBIDO: 1|54|27639
2025-07-17 01:28:30.133501 | 1 | 23 | RECEBIDO: 1|23|58900
2025-07-17 01:28:30.136662 | 1 | 16 | RECEBIDO: 1|16|23053
2025-07-17 01:28:31.826115 | 3 | 76 | RECEBIDO: 3|76|47286
2025-07-17 01:28:31.827065 | 2 | 13 | ENVIADO: 2|13|26429
2025-07-17 01:28:32.984513 | 1 | 76 | RECEBIDO: 1|76|04941

```

Esse padrão se repete para todos os processos, indicando que o coordenador respeita a ordem FIFO (*First-In, First-Out*) da fila interna (*request\_queue* em “invocador.py”) mantendo

estritamente a ordem de chegada das requisições e só concede o *grant* quando a região crítica está livre. Com base nos *timestamps*, verifica-se também, analisando o tempo entre o envio do *grant* e a próxima *release* (tempo de permanência na região crítica), que, além do tempo de permanência estar sendo respeitado, não há concorrência simultânea na região crítica e que o sistema está sincronizado de acordo com os parâmetros de entrada.

Mesmo randomizando os tempos, o tempo total de execução para 5 processos com 3 repetições acabou sendo de aproximadamente 28 segundos, revelando poucos segundos de overhead acumulado, distribuído entre os tempos de comunicação e processamento das mensagens.

Primeiro *timestamp* do “coordenador\_log.txt”:

```
2025-07-17 01:28:30.119990 | 1 | 76 | RECEBIDO: 1|76|89820
```

Último *timestamp*:

```
2025-07-17 01:28:59.528805 | 3 | 16 | RECEBIDO: 3|16|55929
```

Os logs também permitem identificar que há pouca variação no tempo de reposta do coordenador. Após um *release*, o *grant* ao processo agora no topo da fila (desde que há um processo na fila) é enviado rapidamente, não demonstrando custos adicionais de manipulação de fila ou pra sincronização entre threads.

## 4. Conclusão

Pode ser constada corretude do funcionamento por meio da análise dos *timestamps* ordenados e suas respectivas mensagens entre o “coordenador\_log.txt” e o “resultado.txt”. A sequência de mensagens *grant* no log e a sequência de escritas no “resultado.txt” estão perfeitamente alinhadas, tanto em ordem quanto em tempo. Isso demonstra que apenas um processo entra na região crítica por vez, e que o sistema está corretamente sincronizado, pois a alternância de *grant/release* é respeitada e os resultados são consistentes com o controle centralizado e ordem do recebimento das mensagens de *request*. Ou seja, o sistema implementa corretamente o algoritmo centralizado de exclusão mútua, com sincronização garantida pelo coordenador e evidenciada nos arquivos de log e resultado.