12 de outubro de 2021

Universidade Federal do Rio de Janeiro

Sistemas Distribuídos

Trabalho 3

Bruno Dantas de Paiva
Rodrigo Mendes Palmeira

1 Escolhas de Implementação

1.1 Escolha da Linguagem de Programação

Decidimos utilizar a linguagem Go para o projeto por termos um interesse prévio de aprender a linguagem somado ao seu bom balanço de capacidade de realizar ações de baixo nível, necessárias para o projeto, e certas facilidades modernas como garbage collection. Ademais, Go foi feito pensando em sua aplicação em sistemas distribuídos com um tipo de primitiva especial chamada de *Go routines*(Rotinas GO) que possibilitam o uso de CSP(Communicating Sequential Processes)[1] que achamos bastante interessante de utilizar nos trabalhos da disciplina. Além disso o código está disponível no github pelo link `https://github.com/DantasB/Distributed-Systems/tree/main/Trabalho_3`

1.2 Interface de Comunicação entre Processos e Coordenador

Decidimos ter um formato fixo de 32 bits para as mensagens trocadas entre os processos e o coordenador. Os 2 primeiros bits encapsulam o tipo de mensagem: 00 é Erro, 01 é Request, 10 é Grant e 11 é Release. Os demais 30 bits encapsulam o número do processo. Decidimos fazer dessa forma, não utilizando um separador como, por exemplo, | e não sendo uma string, para tornar o payload menor e o protocolo mais eficiente. Para isso, tivemos que definir que a mensagem é transmitida como *big endian*. Como utilizamos a mesma linguaguem(go) no código do coordenador e dos processos, não foi necessário definir outras questões de compatibilidade. As mensagens são armazenadas internamente nos programas utilizando *unsigned* de 32 bits e processadas utilizando máscaras definidas no código através de operações *bit-wise*.

A comunicação entre os processos e o coordenador é feita através de sockets TCP de tal forma que a conexão é aberta pelo processo quando esse decide mandar um Request e encerrada pelo coordenador após esse enviar a mensagem Release. Decidimos utilizar o protocolo TCP, pois ele nos permitiu não nos preocupar a com perda de pacotes e também facilitar o gerenciamento da mensagens pelo coordenador. O fato do coordenador manter

[1] https://www.cs.cmu.edu/ crary/819-f09/Hoare78.pdf

diversas conexões TCPs abertas pode gerar problemas de escalabilidade, porém, o algorítimo centralizado de exclusão mútua distribuída funciona bem quando não há um uso intenso, por parte dos processos, da região critica, assim entendemos que utilizar esse protocolo de transporte é a melhor alternativa por simplificar o código e, ao mesmo tempo, não impactar de forma significativa a performance do caso esperado.

2 Implementação

2.1 Fila thread-safe

Decidimos implementar a fila de processos de tal forma que ela seja *thread-safe*, ou seja, que o restante do código possa chamar seus métodos como *push* ou *pop* sem se preocupar com primitivas de sincronização. Para tal, utilizamos uma slice de go (lista) como a pilha propriamente dita, além de um lock mutex e uma variável de condição para controlar o acesso concorrente à estrutura de dados. Assim, a thread que recebe os pedidos dos processos, a que processa os pedidos e a do terminal usam a mesma pilha sem ter que se preocupar com condições de corrida.

2.2 Arquitetura do Coordenador

O coordenador foi projetado para executar de forma concorrente com 3 rotinas go ou *user-level threads*. A primeira é um loop infinito que aceita uma nova conexão TCP, verifica se é um processo com uma mensagem de Request e, se for o caso, o coloca na fila para ser processado. A segunda é, também, um loop infinito que retira um processo da fila(o primeiro dela no caso), manda uma mensagem Grant para ele e espera a mensagem Release como resposta. A última é a que processa as entradas pelo terminal e, para cada entrada do usuário, verifica se é um dos comandos(Kill, Queue e Amount) previstos e o processa. O primeiro encerra o programa, o segundo imprime a fila de processos e o terceiro retorna o número de vezes que determinado processo recebeu o Grant. Para escrever no arquivo de logs, utilizamos o logger da biblioteca padrão de go que é *thread-safe* e consegue escrever timestamps com precisão de microssegundos.

2.3 Arquitetura dos Processos

O código do processo é relativamente simples. Ele se conecta com o coordenador através de uma conexão TCP, depois, por r repetições: Manda uma mensagem de Request para o servidor indicando que o processo pn requeriu acesso, espera a reposta com uma mensagem Grant do servidor para o processo pn, escreve no arquivo *resultado.txt*, espera k segundos e envia a mensagem Release para o servidor. Pn, k e r são os argumentos de linha de comando passados para o programa.

2.4 Validação dos Logs

Para a validação dos logs também foi criado um script em go para verificar as integridades requisitadas pelo trabalho. Pela própria linguagem possuimos implementações simples que facilitavam o processo de verificação tal como a função *after* da biblioteca *time* que permite com que o usuário avalie se um tempo aconteceu após um outro tempo de uma forma mais simples. Ademais, optamos uma exposição de erros ao usuário que torne possível avaliar em qual arquivo de log o erro foi encontrado.

3 Estudos de Caso

Para cada um dos estudos de caso foram criados scripts em bash para permitir com que estes fossem automatizados mais facilmente. Além disso, para cada log gerado (do coordenador e do cliente) foi executado o nosso código para validar a integridade dos arquivos de log, garantindo que o arquivo gerado estivesse correto.

Além disso, para a geração dos gráficos foi criado um processo auxiliar utilizando a linguagem python.

3.1 Caso 0

Para este caso não foi criado nenhum gráfico por este haver somente um n fixo e servir somente para validação da integridade do código. Deste modo não será disponibilizada nenhuma imagem a respeito deste. Contudo, todo o arquivo de log desta run pode ser encontrado neste link: `https://github.com/DantasB/Distributed-Systems/blob/main/Trabalho_3/Results/Case0/resultado_0.txt`

3.2 Caso 1

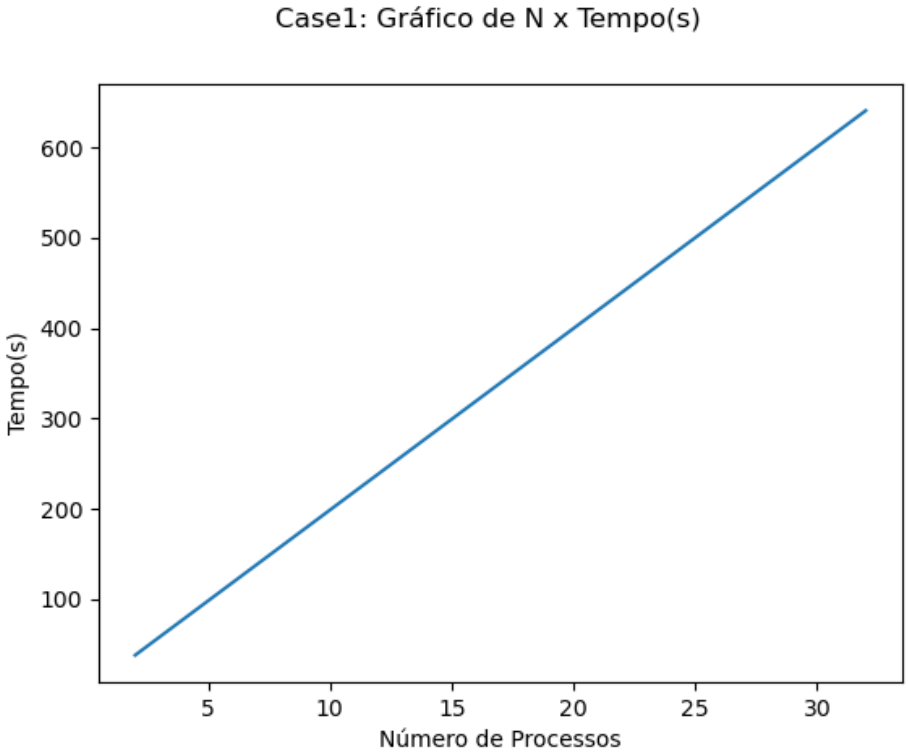

Figura 1: Gráfico de execução do caso 1

Para este caso, é possível observar que conforme o número de processos aumentava, o tempo necessário para a execução do código aumentava de forma linear.

3.3 Caso 2

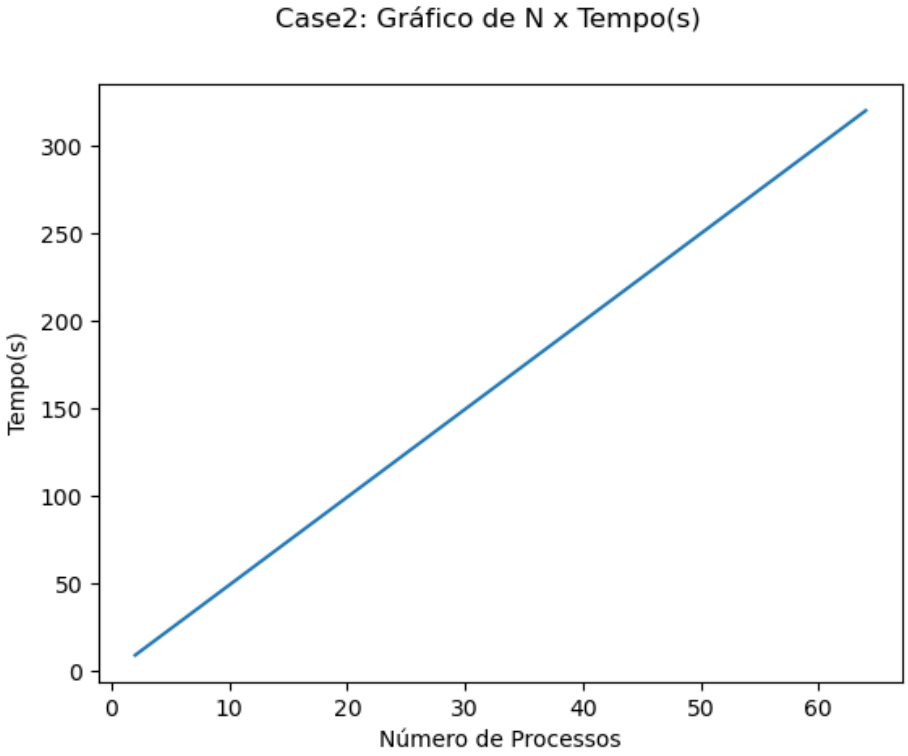

Figura 2: Gráfico de execução do caso 2

Também para este caso, tal como o anterior, o número de processos seguia o mesmo padrão de aumento de tempo linear, conforme o número de processos aumentava.

3.4 Caso 3

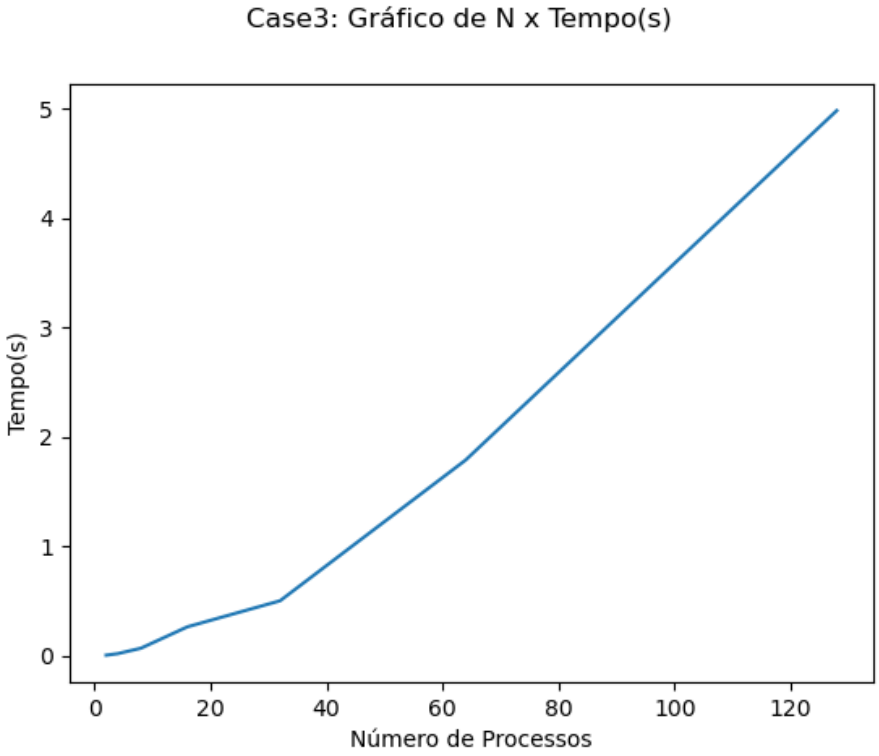

Figura 3: Gráfico de execução do caso 3

Diferentemente dos outros dois casos, este gráfico tem um comportamento não tão linear, onde quando os número de processos são baixos, o tempo tende a crescer em uma taxa menor conforme o número de processos aumenta. Mas, quando o número de processos tende a aumentar em uma escala maior, o tempo aumenta linearmente.

Como era esperado pelo algoritmo centralizado de exclusão mútua distribuída, obtivemos comportamento linear e bons tempos. O tempo mínimo que deveria ser executado é o número de processos (n) * número de repetições (r) * tempo de espera (k), para os casos em que k é igual a 0 qualquer valor acima disso é considerado válido.

Deste modo, ao observar o gráfico, podemos notar que há um baixo overhead, indicando que o sistema está funcionando de forma bastante efi-

ciente. Porém é importante pontuar que esses testes foram executados na mesma máquina, assim não sentimos nos estudos de caso os atrasos provenientes do deslocamento dos pacotes na rede que, em situações reais, tenderão a ser um overhead significativo.