Relatório - Sistemas Distribuídos

Difusão Atômica utilizando Mecanismo de Ordenação Total Baseado em Privilégio e Algoritmo de Exclusão Mútua Distribuída Baseado em Anel

Christiane Fernandes Peressutti
Engenharia de Computação
Universidade Federal de Santa Catarina - UFSC
Email: christiane.peressutti@grad.ufsc.br

***Resumo*—Serão tratadas duas implementações relacionadas a sistemas distribuídos, sendo estas de difusão atômica com mecanismo de ordenação total baseado em privilégio e algoritmo de exclusão mútua distribuído baseado em anel, sendo ambas representadas a seguir em códigos com saídas intuitivas quanto a seu funcionamento.**

I. Introdução

Em sistemas distribuídos, diversos conceitos são abordados quanto a processos, comunicação entre processos distribuídos, concorrência e sincronização, tolerância a faltas, segurança e seus estudos de caso.

Serão expostas implementações e conceitos quanto a difusão atômica utilizando o mecanismo de ordenação total baseado em privilégio e ao algoritmo de exclusão mútua distribuído baseado em anel.

Visto que suas representações poderão ser de forma gráfica ou escrita, a linguagem de programação escolhida foi Python.

II. Difusão Atômica com Mecanismo de Ordenação Total Baseada em Privilégio

A. Conceito

O funcionamento do mecanismo de ordenação total baseada em privilégio divide-se em duas partes prinicpais, sendo os remetentes (emissores) e os destinatários (receptores).

Funciona da seguinte forma: os emissores formam um anel virtual no qual circula um token, cujo emissor que estiver de posse deste tem o privilégio de difundir suas mensagens [1], como ilustrado abaixo. As mensagens enviadas tem mesmo conteúdo e ordem de chegada.

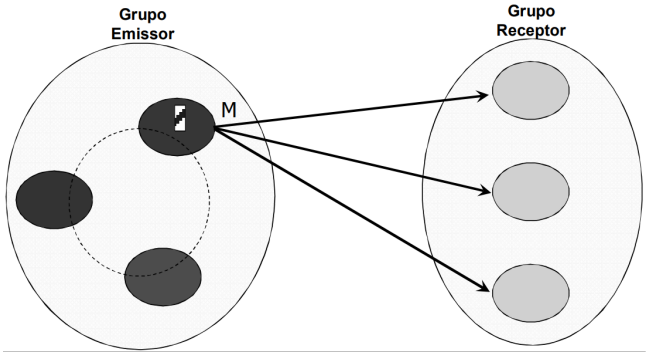

Figura 1: Funcionamento do mecanismo de ordenação total baseado em privilégio

Por parte dos emissores, o algoritmo é inicializado com um conjunto vazio. Para o primeiro processo inicializa-se o token com 1 e executa-se a primitiva que enviará o token para o primeiro processo. Em seguida, executa-se a difusão de ordem total sobre a mensagem, que é unida ao conjunto. Ao receber o novo token, cada mensagem dentro do conjunto executa a primitiva de envio com o valor atual do token, que irá rotacionar, para os receptores. O token é enviado para o próximo processo após ser feito um cálculo modular que itera pelos processos participantes de forma anelar, que ao receber executa novamente o algoritmo [2].

Já o algoritmo do receptor inicializa os conjunto de dados de próxima entrega com todos os valores igual à 1 e fim como vazio. Em seguida, começa a processar continuamente o recebimento de mensagens, cada uma com seu número de sequência. Logo após, implementa a união de novas mensagens no conjunto de pendências em laço repetitivo, onde se verifica a existência de uma mensagem com número de sequência pertencente ao conjunto de pendências. Caso sim, o número de sequência receberá o valor da próxima entrega. Em sequência, executa a primitiva de entrega [2].

Esta técnica pode apresentar alguns problemas, sendo a perda do token caso haja falha no processo que o esteja usando [1].

B. Implementação

Dada a descrição do algoritmo anteriormente, abaixo encontra-se o código implementado para demonstração do funcionamento. O número de processos do remetente pode ser alterado na chamada da linha 7 `numproc = 3`.

```
1   from emissor import process
2   from receptor import process_pi
3   from comunica import send
4
5   def main():
6       i=0
7       numproc = 3
8
9       if i <= numproc:
10              r = process(numproc)
11              e = process_pi()
12
13  if __name__ == '__main__':
14      main()
```

Listing 1: Código main

```
    from receptor import receive
    from comunica import send

    def TO_broadcast(m):
        global tosendsi
        tosendsi.append(m)

    def process(si):
        global token, seqnum
        global tosendsi
        token = 'teste'
        seqnum = 0
        destino = "127.0.0.1"
        tosendsi = []
        s1 = 1

        if si == s1:
            seqnum +=1
            send(token, s1, destino)
            print('enviado')
            si += 1

        boole = True
        while boole:
            receive(token, seqnum)
            for m_prime in tosendsi:

                send(m_prime, token.seqnum, destino)
                print('enviado')
                #token(seqnum := token.seqnum+1)
                seqnum +=1
                si += 1

            tosendsi = []

            send(token, (si+1), destino)
            print('enviado')

            parar = 1
            if parar >= 1:
                #time.sleep(0.5)
                boole = False
```

Listing 2: Código do Emissor

```
    nextdeliverpi = 0
    pendingpi = []

    def receive(m, seqnum):
        pendingpi.append((m, seqnum))
        print("todas as mensagens recebidas: ",
    pendingpi)

    def deliver(m):
        print("mensagem recebida: ", m)

    def process_pi():
        global nextdeliverpi
        boole = True
        while boole:
            for i, (m, seqnum) in enumerate(
    pendingpi):
                if seqnum == nextdeliverpi:
                    deliver(m)
                    nextdeliverpi += 1
                    pendingpi.pop(i)
                    break
```

Listing 3: Código do Receptor

```
    import socket

    def send(message, seqnum, destination):
        # Cria um socket UDP
        with socket.socket(socket.AF_INET, socket.
    SOCK_DGRAM) as s:
            m = message.encode()

            # Envia a mensagem para o destino
            s.sendto(m, (destination, 5000))
```

Listing 4: Código de Envio de Mensagem

III. Algoritmo de Exclusão Mútua Distribuído Baseado em Anel

A. *Conceito*

O algoritmo de exclusão mútua distribuído baseado em anel (token ring) é formado por um grupo não ordenado de processos. É embasado na topologia lógica de anel construído em software. Nesse anel, cada processo, antes encontrados em uma rede de barramento, será designado a uma posição, que pode ser alocado em ordem numérica de endereços de rede ou outro meio [3].

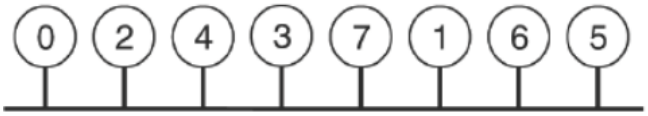

Figura 2: Processos na rede desordenados

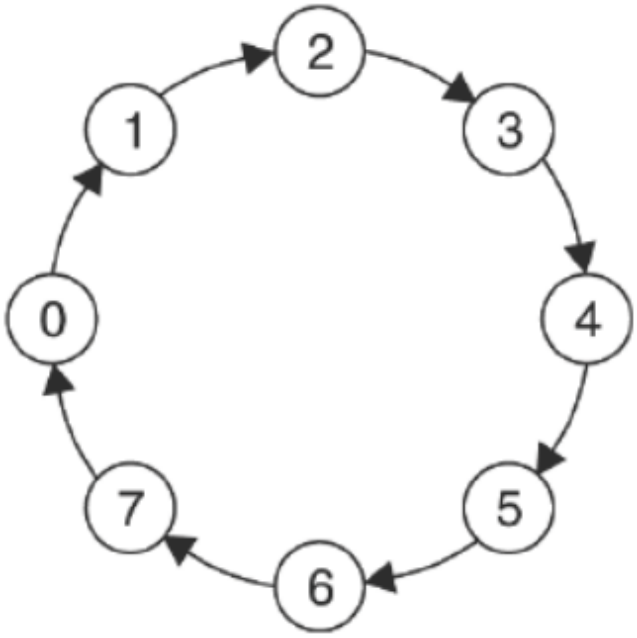

Figura 3: Anel lógico inicializado

Quando o anel é inicializado, o processo 0 recebe uma ficha (token), que circula ao redor do anel. Quando um processo adquire o token de seu vizinho, ele verifica se está tentando entrar em uma região crítica. Nesse caso, o processo entra na região, faz todo o trabalho necessário e sai da região. Depois de sair, ele passa o token para o próximo processo no anel.

Não é permitido entrar novamente na região crítica utilizando o mesmo token. Se um processo recebe o token de seu vizinho e não está interessado em entrar em uma região crítica, ele apenas passa o token para o próximo processo. [4]

Apenas um processo pode obter o recurso por vez, evitando assim inanição. Todavia, se o token for perdido, precisará ser regenerado, considerando que a quantidade de tempo entre aparições sucessivas do token na rede é ilimitado. Ademais, se um processo cair, poderá ser removido do grupo, passando o token para o próximo. Para isso, todos precisam manter a configuração corrente do anel. [4]

B. Implementação

Como no tópico visto precedentemente, abaixo encontra-se o código implementado para demonstração do funcionamento. O número de nodos pode ser alterado na chamada da linha 41 `num_nodes = 7`.

```
1  import threading
2
3  class Token: #token que e passado entre os nos do anel
4      def __init__(self, value):
5          self.value = value
6
7  class Node(threading.Thread): #representa cada no do anel. Cada no e executado em uma thread separada e possui um identificador, um token e uma referencia para o proximo no no anel
8      def __init__(self, id):
9          super().__init__()
10         self.id = id
11         self.token = None
12         self.next_node = None
13
14     def run(self): #verifica se possui o token. Se possuir, entra na secao critica e imprime uma mensagem. Caso contrario, passa o token para o proximo no
15         while True:
16             if self.token is not None:
17                 if self.token.value == self.id:
18                     print(f"Nodo {self.id} na secao critica com {self.token}\n")
19                     self.token = None
20                     print(f"Nodo {self.id} agora com {self.token}\n")
21                 else:
22                     self.pass_token()
23             #else:
24             #    self.receive_token(self.next_node.token)
25
26     def pass_token(self): #passa o token para o proximo no
27         self.next_node.receive_token(self.token)
28         print(f"Seguiu para o proximo {self.next_node.id} com token: \n", self.token)
29         self.token = None
30
31     def receive_token(self, token): #recebe o token de um no anterior
32         if token is not None:
33             self.token = token
34             print("Recebeu token: \n", token)
35
36     def set_next_node(self, next_node): #seta para o proximo nodo
37         self.next_node = next_node
38         print("Proximo: \n", next_node)
39
40 def main(): #sao criados os nos e configuradas as referencias para o proximo no. Em seguida, cada no e iniciado em uma thread separada
41     num_nodes = 7
42     nodes = []
43
44     for i in range(num_nodes):
45         #next_node = nodes[i].set_next_node(nodes[(i + 1) % num_nodes])
46         node = Node(i)
47         nodes.append(node)
48
49     for i in range(num_nodes):
50         nodes[i].set_next_node(nodes[(i + 1) % num_nodes])
51
52
53     for node in nodes:
54         node.token = Token(node.id) # define o token inicial para cada no
55         threading.Thread(target=node.run).start()
56
57 if __name__ == "__main__":
58     main()
```

Listing 5: Código de implementação do Algoritmo de Exclusão Mútua Distribuído Baseado em Anel

Execução dos Códigos

Para executar os códigos apresentados será preciso instalar uma IDE da preferência do leitor, com indicação para VSCode, Spyder ou IDLE (este instalado junto ao Python).

Fazer a instalação do Python de versão posterior a 3.6. Ao abrir o executável de instalação selecionar a opção para incluir ao PATH.

Abrir a pasta contendo o código na IDE e selecionar a opção de executar, executar e depurar ou utilizar o comando: `python main.py`

Caso escolha pelo VSCode, antes do passo anterior, com o comando `crtl + shift + P` selecionar o interpretador correspondente a versão do Python.

IV. Considerações Finais

Neste relatório foram apresentados conceitos e funcionamento dos algoritmos, bem como suas implementações Todo código está disponível no repositório. Podem ocorrer alguns problemas na execução do código baseado em privilégio que até o fim deste relatório não foram solucionados, de forma que precisarão ser resolvidos adiante. Além disto, a implementação de saída gráfica não fora possível por conta de problemas de renderização.

Referências

[1] P. L. C. Lung, "Comunicação de grupo: Disfusão confiável e atômica," https://www.inf.ufsc.br/~frank.siqueira/INE5418/Lau/1-%20ComGrupo.pdf, 2011.

[2] X. Défago, A. Schiper, and P. Urbán, "Totally ordered broadcast and multicast algorithms: A comprehensive survey," 2000.

[3] P. A. Nascimento, "Sistemas distribuídos, capítulo - sincronização," http://profs.ic.uff.br/~simone/sd/contaulas/aula14.pdf, 2019.

[4] A. S. Tanenbaum and M. v. Steen, *Sistemas Distribuídos: princípios e paradigmas*. São Paulo: Person Prentice Hall, 2007.