Gabriel Medeiros Lopes Carneiro
Paulo Arthur Sens Coelho
Erik Kazuo Sugawara

Relatório

Florianópolis, SC
2022

Gabriel Medeiros Lopes Carneiro
Paulo Arthur Sens Coelho
Erik Kazuo Sugawara

Relatório

Trabalho apresentado para avaliação de uma implementação de uma biblioteca para comunicação confiável e algoritmo distribuído, na disciplina INE 5418 - Computação Distribuída, sob Orientação do Prof. Dr. Odorico Machado Mendizabal

Universidade Federal de Santa Catarina
Departamento de Informática e Estatística
Programa de Graduação

Orientador: Odorico Machado Mendizabal

Florianópolis, SC
2022

Gabriel Medeiros Lopes Carneiro
Paulo Arthur Sens Coelho
Erik Kazuo Sugawara
Relatório/ Gabriel Medeiros Lopes Carneiro
Paulo Arthur Sens Coelho
Erik Kazuo Sugawara. – Florianópolis, SC, 2022-
?? p. : il. (algumas color.) ; 30 cm.

Orientador: Odorico Machado Mendizabal

Relatório (Graduação) – Universidade Federal de Santa Catarina
Departamento de Informática e Estatística
Programa de Graduação, 2022.

1. Computação Distribuída. 2. Sockets. 3. Ordem Causal. 4. Ordem Total. 5. Biblioteca.

CDU 02:141:005.7

Gabriel Medeiros Lopes Carneiro
Paulo Arthur Sens Coelho
Erik Kazuo Sugawara

Relatório

Trabalho apresentado para avaliação de uma implementação de uma biblioteca para comunicação confiável e algoritmo distribuído, na disciplina INE 5418 - Computação Distribuída, sob Orientação do Prof. Dr. Odorico Machado Mendizabal

---

Odorico Machado Mendizabal
Orientador

Florianópolis, SC
2022

Lista de ilustrações

Sumário

Identificação

Universidade Federal de Santa Catarina
Departamento de Informática e Estatística
Ciências da Computação
INE 5425 - Modelagem e Simulação

Contato

Gabriel Medeiros Lopes Carneiro
Email: gabriel.mlc@grad.ufsc.br

Paulo Arthur Sens Coelho
Email: paulo.sens.coelho@grad.ufsc.br

Erik Kazuo Sugawara
Email: erik.sugawara@grad.ufsc.br

Parte I

Biblioteca de Comunicação Confiável

1 Sobre o Trabalho

1.1 Introdução

Um sistema distribuído é caracterizado por componentes localizados em uma rede de computadores, que se comunicam e coordenam suas ações através de troca de mensagens e que podem sofrer influências internas ou externas, as quais podem prejudicar a sua interoperabilidade. Através da concorrência entre os componentes comunicantes, o estado de seus dados se tornam inconsistentes a medida que não existe uma sincronização. Seja por conta da ausência de um relógio global ou de uma memória compartilhada, se torna impossível realizar uma sincronização perfeita entre as trocas de mensagens. Além disso, em seu aspecto físico, componentes são suscetíveis a falha e podem impedir o funcionamento do sistema. Sendo assim, o objetivo deste trabalho é criar uma biblioteca de comunicação confiável que seja capaz de garantir a ordem no recebimento e envio de mensagens seguindo critérios de ordem causal e total, de modo que os participantes da comunicação consigam estabelecer troca de mensagens do tipo 1 : 1 e 1 : n.

1.2 Metodologia

Para realizar a criação da biblioteca foi escolhida a linguagem Python por sua versatilidade e legibilidade. Em conjunto, foi usado a biblioteca de Multiprocessos para permitir que os processos que instanciam os sockets sejam possíveis de serem executados em paralelo, garantindo o recebimento e envio de mensagens de múltiplos nodos. Cada nodo contém dois processos que instanciam sockets: o primeiro é utilizado para o recebimento de mensagens e o segundo é utilizado para enviar mensagens, uma vez que não conseguimos reutilizar o mesmo socket para realizar operações de envio e recebimento. Para garantir a entrega dos dados, foi utilizado o Protocolo TCP. No socket foi utilizado o domínio AF_INET para criarmos um socket capaz utilizar os endereços com seus respectivos Ip e Porta, dessa forma os nodos são capazes de comunicar entre si. Além disso, a garantia do recebimento de pacotes ponto a ponto facilita a elaboração de uma biblioteca mais robusta.

2 Desenvolvimento

2.1 Biblioteca

2.1.1 Nodo

A estrutura de cada nodo é composta por três tipos de buffer: do processo, de entrada e de saída. O de processo é responsável por armazenar as mensagens que precisam aguardar outras chegarem. O buffer de entrada armazena os IDs de cada mensagens e verifica se a mensagem já pode ser entregue. O buffer de saída adiciona os IDs em cada mensagem para cada um dos endereços de nodo. Para comunicação são utilizados dois sockets: um para recebimento de mensagens e outro para envios. Cada socket tem o seu próprio endereço, dessa forma um nodo tem dois sockets com diferentes portas. Na figura **??**, podemos ver o esquema de um nodo na rede de comunicação.

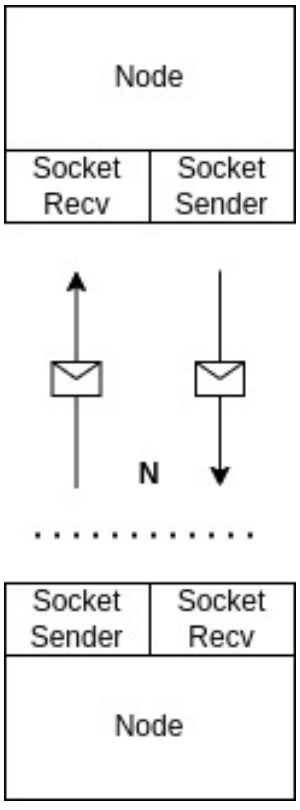

Figura 1 – Estrutura de um nodo

2.2 Mensagem

Internamente as mensagens são compostas por quatro atributos: dados a serem enviados, id da mensagem, id do processo e endereço do processo. Cada mensagem é

transferido na forma de bytes, onde é realizado um parse dela. Em nossa biblioteca, enviamos uma string com o seguinte corpo: DataTxt # MsgID # ProcessID # ProcessAddress.

2.2.1 Recebimento de mensagens

O método responsável por receber as mensagens verifica se a mensagem recebida está no buffer do processo. Se sim, ignora a mensagem. Do contrário, verifica se ela está no buffer de entrada e se já pode ser entregue. Se a mensagem está em seu estado válido, então a mensagem é entregue.

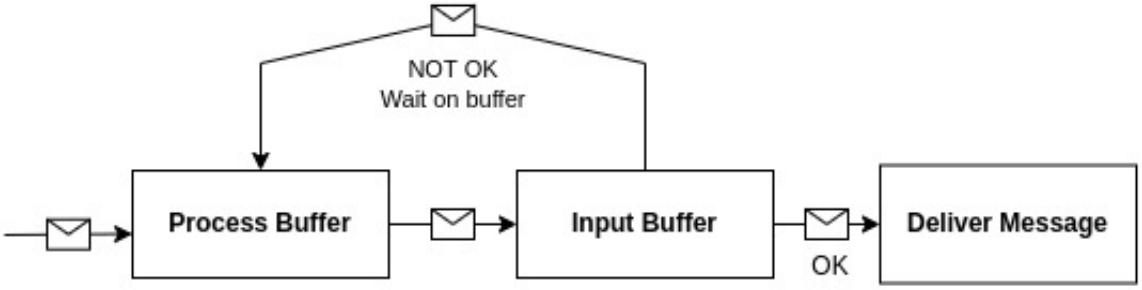

Figura 2 – Estrutura do recebimento de mensagens.

2.2.2 Envio de mensagens

Para enviar as mensagens é necessário adicionar o ID em cada uma delas antes do envio. Sendo assim, foi utilizado um buffer de saída que armazena os IDs de cada mensagem para cada um dos comunicantes.

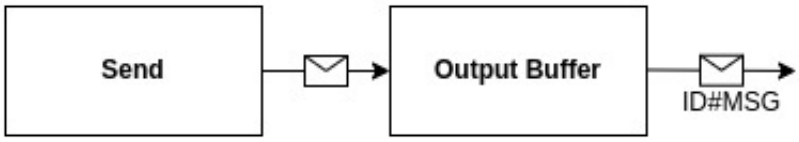

Figura 3 – Estrutura do envio de mensagens.

2.3 Ordenação Causal

Para garantia da ordem de mensagens de 1 para 1, foi elaborado no nodo um esquema de buffers onde uma mensagem que chega no nodo é verificada se já foi recebida ou está aguardando outras mensagens, caso ela já foi recebida essa mensagem é ignorada, do contrário aguarda pelas restantes e é enviada posteriormente.

2.4 Ordenação Total

Para assegurar que todas as mensagens sejam entregues a todos os nodos pela mesma ordem, foi necessário criar a classe Sequencer que realiza a ordenação das mensagens e envia para todos na rede. Ela é composta por um socket que recebe as mensagens, um buffer de mensagens e uma lista de endereços de cada nodo. Com essas informações é atribuído um número de sequência único para todas as mensagens que são enviadas posteriormente.

2.5 Algoritmo Token Ring

Nesse algoritmo é construído um anel lógico com os processos comunicantes. Em sua inicialização o processo 0 recebe um token. Ele deve circular pelo anel passando de um processo k para o k + 1. Quando o processo recebe o token ele pode executar a região crítica (uma única vez) e, após sair, passa o token para o próximo processo. Se o processo não desejar entrar na região crítica, deve apenas passar o token para o próximo processo.

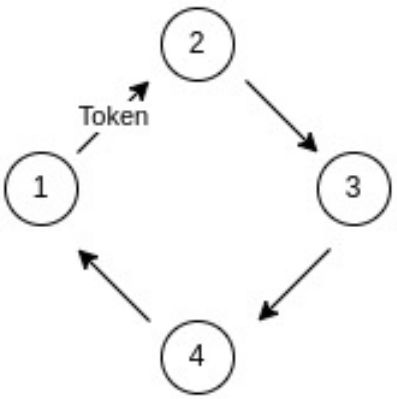

Figura 4 – Esquema de um token ring.

2.6 Execução

Para iniciar os procedimentos, foi criado um arquivo de teste chamado "config_node_test.txt", onde podemos configurar as mensagens que cada processo irá trocar entre si. Note que não pode existir espaços na mensagem que está sendo enviada e o número dos nodos não pode exceder o tamanho definido na constante "NUMBER_OF_PROCESS"no arquivo "utils.py". Além disso, foi adicionado o método sleep() para simular o processamento e atraso das mensagens.

2.6.1 1 para 1

A configuração para enviar de um para um é feita da seguinte forma: ID Processo Origem : Mensagem : ID Processo Destino. Por exemplo, se tivéssemos a seguinte configuração: "0 : OlaMundo : 1", o nodo 0 irá enviar uma mensagem OlaMundo para o nodo 1.

2.6.2 1 para N

A configuração para realizar o *broadcast* é feita da seguinte forma: "ID Processo Origem : Mensagem : -1", onde o -1 indica que será feito um broadcast. Por exemplo, a seguinte linha "2 : Teste : -1", fará com que o nodo 2 envie a mensagem "Teste"para todos os nodos.

2.6.3 Token

Para testar o funcionamento do token podemos realizar a seguinte configuração: "ID Processo Origem : token : ID Processo Destino", onde token é uma palavra reservada. Se utilizarmo a seguinte linha "0 : token : 3", iremos observar o token indo do nodo 0 até o nodo 3. Note que não podemos adicionar a nomenclatura broadcast -1 no ID de destino.

2.7 Resultados

Para verificar o funcionamento básico das mensagens de um processo para outro, foi realizado um teste onde o nodo 1 envia duas mensagens para o nodo 2. Na figura **??**, é mostrado a saída indicando que a mensagem foi entregue corretamente.

```
1: enviou 'hi' para '2'.
1: enviou 'hi2' para '2'.
2: mensagem 'hi', recebida de '1', aguardando para ser entregue.
2: mensagem 'hi', recebida de '1', entregue.
2: mensagem 'hi2', recebida de '1', aguardando para ser entregue.
2: mensagem 'hi2', recebida de '1', entregue.
```

Figura 5 – Envio de mensagens sendo entregue entre os processos.

Para verificar o funcionamento do broadcast, fizemos com que o nodo 2 realizasse este processo. Na figura **??**, é demonstrado o nodo 2 realizando um broadcast.

Para verificar o token ring, iniciamos o nodo 0 com um token.Na figura **??** conseguimos verificar a transferência de token entre os nodos na rede.

```
2: enviou 'TesteBroadcast' para '3'.
3: mensagem 'TesteBroadcast', recebida de '2', aguardando para ser enviada aos outros nós.
3: enviou 'TesteBroadcast' para '0'.
3: enviou 'TesteBroadcast' para '1'.
0: mensagem 'TesteBroadcast', recebida de '3', aguardando para ser entregue.
0: mensagem 'TesteBroadcast', recebida de '3', entregue.
1: mensagem 'TesteBroadcast', recebida de '3', aguardando para ser entregue.
1: mensagem 'TesteBroadcast', recebida de '3', entregue.
```

Figura 6 – Broadcast feito pelo nodo 2.

```
0: enviou 'token' para '2'.
2: enviou 'token' para '0'.
0: enviou 'token' para '1'.
1: enviou 'token' para '2'.
2: enviou 'token' para '0'.
0: enviou 'token' para '1'.
1: enviou 'token' para '2'.
Finalizando processo '0 - Receiver'
Finalizando processo '1 - Receiver'
Finalizando processo '2 - Receiver'
Finalizando processo 'Sequencer receiver'
Todas as mensagens foram enviadas.
```

Figura 7 – Token iniciado no nodo 0 e sendo passado para os demais.

2.8 Conclusão

Os sistemas distribuídos possuem componentes que realizam troca de mensagens entre si. É necessário uma sincronização para que seja possível coordenar as suas ações, uma vez que pode existir concorrência no envio de mensagens entre os comunicantes. Para resolver esse problema, foi elaborado uma biblioteca capaz de garantir o ordenamento e recebimento correto das mensagens. Durante a sua elaboração, foi necessário compreender o funcionamento da ordem causal e total no recebimento de mensagens, como também entender os efeitos do algoritmo de exclusão mútua chamado *token ring* dentro da rede. Dessa forma, conseguimos compreender os problemas de sincronização que envolvem concorrência e das dificuldades de criar uma biblioteca capaz de garantir a ordem e integridade das mensagens.

ANEXO A – Biblioteca

A.1 Sequencer

```
from socket import socket, AF_INET, SOCK_STREAM
from typing import List

from .utils import Address, parse_msg, Message, Buffer, address_to_id


class Sequencer:
    def __init__(self, id_: int,
                 addresses: List[Address],
                 address: Address):
        self._seq_num: int = 1
        self.addresses: List[Address] = addresses
        self.address: Address = address
        self._sends_messages: Buffer = {address: [] for address in
   addresses}
        self._receiver_socket: socket = socket(AF_INET, SOCK_STREAM)
        self._receiver_socket.bind(address)
        self.id = id_

    def on_recv(self, msg: bytes) -> None:
        message: Message = parse_msg(msg)
        process_address = message.origin_address

        if (message in self._sends_messages[process_address]):
            print(f"{self.id}: J havia recebido '{message.data}',
   ignorando...")
            return None

        print(f"{self.id}: mensagem '{message.data}', "
              f"recebida de '{message.origin_id}', aguardando para ser
   enviada aos outros ns.")

        self._sends_messages[process_address].append(message)
```

```
        broadcast_message: Message = Message(data=message.data, id_=self.
    _seq_num,
                                             origin_id=self.id,
    origin_address=self.address)
        self._seq_num += 1
        self.send(message=broadcast_message, origin_address=
    process_address)

        return None

    def start_socket(self):
        self._receiver_socket.listen(len(self.addresses))
        while True:
            conn, addr = self._receiver_socket.accept()
            with conn:
                data = conn.recv(1024)
                self.on_recv(msg=data)
                conn.sendall(data)

    def send(self, message: Message, origin_address: Address) -> None:
        for address in self.addresses:
            if (origin_address != address) and (address != self.address):
                self.send_to_socket(message=message, address=address)

        return None

    def send_to_socket(self, address: Address, message: Message):
        s = socket(AF_INET, SOCK_STREAM)
        s.connect(address)
        s.sendall(message.to_bytes())
        s.close()
        print(f"{self.id}: enviou '{message.data}' para '{address_to_id(
    address)}'.")
        return None
```

A.2 Utils

```
from __future__ import annotations

from typing import Tuple, Dict, List

HOST: str = "127.0.0.1"
PORT: int = 5000
TOKEN: str = "token"
NUMBER_OF_PROCESS: int = 3
ADDRESS_TO_ID: Dict[Address, int] = {}


class Message:
    def __init__(self, data: str, id_: int, origin_id: int,
   origin_address: Address):
        self.data: str = data
        self.id: int = id_
        self.origin_id: int = origin_id
        self.origin_address: Address = origin_address

    def __eq__(self, other: Message) -> bool:
        return (self.data == other.data) and (self.id == other.id) \
               and (self.origin_id == other.origin_id)

    def to_bytes(self) -> bytes:
        return f"{self.data}#{self.id}#{self.origin_id}#{self.
   origin_address}".encode()


def parse_msg(data: bytes) -> Message:
    """
    Example: MsgTxt # MsgID # ProcID # ProcAddress
    """

    data = data.decode()
    splitted = data.split("#")
    msg = splitted[0]
    msg_id = splitted[1]
    proc_id = splitted[2]
    proc_address = splitted[3]
```

```
    return Message(data=msg, id_=int(msg_id),
                   origin_id=int(proc_id), origin_address=eval(
   proc_address))


def parser(file_path: str) -> List[Tuple[int, str, int]]:
    with open(file_path, "r") as file:
        lines = file.readlines()
        messages: List[Tuple[int, str, int]] = []

        for line in lines:
            splitted = line.split(":")

            origin_id: int = int(splitted[0])
            if (origin_id < 0 or origin_id >= NUMBER_OF_PROCESS):
                print(f"ID de origem {origin_id}  invlido. \n"
                      f"Deve ser um valor entre [0, {NUMBER_OF_PROCESS -
   1}]")
                exit()

            message: str = splitted[1].replace(" ", "")

            destiny_id: int = int(splitted[2])
            if (destiny_id < -1 or destiny_id >= NUMBER_OF_PROCESS):
                print(f"ID de destino {destiny_id}  invlido. \n"
                      f"Deve ser um valor entre [-1, {NUMBER_OF_PROCESS -
    1}]")
                exit()

            messages.append((origin_id, message, destiny_id))

    return messages


def address_to_id(address: Address) -> int:
    port: int = address[1]
    port -= PORT
    return port
```

```
Address = Tuple[str, int]
Buffer = Dict[Address, List[Message]]
```