Universidade Federal de Ouro Preto
Instituto de Ciências Exatas e Aplicadas

Trabalho 2: Implementação do algoritmo de berkeley e eleição do anel

Professor: Theo Silva Lins
Aluno: Marcos Henrique Santos Cunha - 19.1.8003

João Monlevade, 7 de Janeiro de 2022.

Introdução

Neste trabalho, será implementado o algoritmo de berkeley e o algoritmo de eleição do anel em um sistema distribuído, com no mínimo 4 máquinas executando. O algoritmo de berkeley é um algoritmo muito efetivo para sincronização de relógios de forma distribuída, e o algoritmo de eleição do anel consiste em eleger um mestre constantemente, de forma que caso o mestre pare de funcionar, outra máquina assuma o posto. Todo o código implementado pode ser encontrado no repositório https://github.com/Marcoshsc/trabalho2-sistemas-distribuidos.

Implementação

A implementação foi feita em duas partes, uma para o algoritmo de berkeley e outra para o algoritmo de eleição do anel. A linguagem escolhida para a implementação foi Python, e ela foi usada para implementar as duas funcionalidades.

Algoritmo de berkeley

A implementação do algoritmo de berkeley é dividida em duas partes: servidor (master) e cliente (slaves). O master é aquele que controla a sincronização dos relógios dos clientes. Inicialmente o código funciona da seguinte maneira:

```
if __name__ == '__main__':

    initiateClockServer(port = 8080)
```

Isto inicializa o servidor. Ao olhar mais de perto a função:

```
def initiateClockServer(port = 8080):

    master_server = socket.socket()
    master_server.setsockopt(socket.SOL_SOCKET,
                                    socket.SO_REUSEADDR, 1)

    print("Socket no node mestre inicializado.\n")

    master_server.bind(('', port))

    # Start listening to requests
    master_server.listen(10)
    print("Servidor de relógio iniciado ... \n")

    # start making connections
    print("Começando a fazer conexões ... \n")
    master_thread = threading.Thread(
                        target = startConnecting,
                        args = (master_server, ))
    master_thread.start()

    # start synchroniztion
    print("Começando a sincronização paralelamente ... \n")
    sync_thread = threading.Thread(
                        target = synchronizeAllClocks,
                        args = ())
    sync_thread.start()
```

De forma simples, o socket é iniciado, para escutar opor requisições, com uma thread sendo disparada para tratar as requisições via rede, permitindo então conexões. Ao olhar mais de perto para a thread criada:

```
def synchronizeAllClocks():

    while True:

        print("Novo ciclo de sincronização iniciado.")
        print("Número de clientes a serem sincronizados: " + \
                                        str(len(client_data)))

        if len(client_data) > 0:

            average_clock_difference = getAverageClockDiff()

            for client_addr, client in client_data.items():
                try:
                    synchronized_time = \
                        datetime.datetime.now() + \
                                        average_clock_difference

                    client['connector'].send(str(
                                synchronized_time).encode())

                except Exception as e:
                    print("Algo deu errado enquanto " + \
                          "enviava tempo sincronizado " + \
                          "para " + str(client_addr))

        else :
            print("Sem dados do cliente.." + \
                        " Sincronização não aplicável.")

        print("\n\n")

        time.sleep(5)
```

A thread consiste de uma função que é responsável por sincronizar o tempo dos clientes conectados. Ela passa pela lista de clientes, calcula a diferença média entre os relógios dos clientes, e envia o tempo sincronizado para todos os outros. A conexão é feita por meio de uma estrutura de lista, onde são armazenados os dados dos clientes que foram conectados, como segue abaixo:

```
def startConnecting(master_server):


    while True:

        master_slave_connector, addr = master_server.accept()
        slave_address = str(addr[0]) + ":" + str(addr[1])

        print(slave_address + " foi conectado com sucesso")

        current_thread = threading.Thread(
                         target = startReceivingClockTime,
                         args = (master_slave_connector,
                                 slave_address, ))
        current_thread.start()
```

Essa thread é disparada para receber as conexões dos slaves, e redireciona para uma outra função de input, para receber o relógio dos slaves também:

```
def startReceivingClockTime(connector, address):

    while True:

        clock_time_string = connector.recv(1024).decode()
        clock_time = parser.parse(clock_time_string)
        clock_time_diff = datetime.datetime.now() - \
                                            clock_time

        client_data[address] = {
                        "clock_time"      : clock_time,
                        "time_difference" : clock_time_diff,
                        "connector"       : connector
                        }

        print("Dados do cliente atualizados com: "+ str(address),
                                            end = "\n\n")
        time.sleep(5)
```

E assim é o funcionamento do node mestre. Agora falando sobre o cliente:

```
def startSendingTime(slave_client):

    while True:
        # provide server with clock time at the client
        slave_client.send(str(
                        datetime.datetime.now()).encode())

        print("Tempo recente recebido com sucesso",
                                            end = "\n\n")
        time.sleep(5)


def startReceivingTime(slave_client):

    while True:

        Synchronized_time = parser.parse(
                        slave_client.recv(1024).decode())

        print("Tempo sincronizado no cliente é: " + \
                                    str(Synchronized_time),
                                    end = "\n\n")
```

Essas duas funções servem para enviar e receber dados de tempo, de forma que o servidor possa receber e enviar dados para o cliente. A parte de inicialização é tal como o node master, somente conecta com o master e fica enviando e recebendo informações de tempo:

```
def initiateSlaveClient(port = 8080):

    slave_client = socket.socket()


    slave_client.connect(('127.0.0.1', port))


    print("Começando a receber dados do servidor\n")
    send_time_thread = threading.Thread(
                       target = startSendingTime,
                       args = (slave_client, ))
    send_time_thread.start()



    print("Começando a receber tempo " + \
                         "sincronizado do servidor.\n")
    receive_time_thread = threading.Thread(
                        target = startReceivingTime,
                        args = (slave_client, ))
    receive_time_thread.start()


if __name__ == '__main__':

    initiateSlaveClient(port = 8080)
```

Algoritmo de eleição do anel

Este algoritmo também é dividido em duas implementações, a do cliente e a do servidor, primeiro será mostrado o servidor:

```python
while True:

    try:
        connection, addr = s.accept()

        process_sockets_list.append(connection)

        recv_process_id = connection.recv(1024)
        from_to_process = recv_process_id.decode('utf-8')

        process_list.append(from_to_process)
        print("Processo: " + from_to_process)

        start_thread = threading.Thread(target=recv_message, args=(connection,))
        start_thread.start()
    except socket.error as msg:
        print("thread failed"+msg)
```

Essa é a linha de execução do servidor, basicamente ele fica escutando por conexões de outros processos numa thread, que roda a seguinte função:

```
def recv_message(conn):
    while True:

        try:
            received = conn.recv(1024)
            msg_token = received.decode('utf-8')
            print("Token recebida: " + msg_token)
        except:
            continue

        if "Coordenador: " in msg_token :
            le=msg_token.split()
            leader=le[1]

        process_index = process_sockets_list.index(conn)

        if len(process_sockets_list)==process_index+1 :
            to_process=0
        else :
            to_process=process_index+1

        try :
            process_sockets_list[to_process].send(received)

            print("Enviando :" + received.decode('utf-8'))

        except :

            if process_list[to_process]≠leader :
                process_sockets_list[to_process+1].send(received)
                print("Enviando :" + received.decode('utf-8'))

            process_sockets_list[to_process].close()

            process_sockets_list.remove(process_sockets_list[to_process])

            process_list.remove(process_list[to_process])
            continue
```

Como mostrado acima, a função executada na thread é responsável por executar a regra de negócio do algoritmo, basicamente trocando tokens entre os nós conectados, de forma que possa ser calculado o melhor elemento para ser eleito. Em caso de um nó desconectar da rede, mesmo que ele seja o eleito mestre, outra eleição será feita e outro nó será escolhido para ser o mestre.

No topo do arquivo, o servidor é colocado numa porta na rede, e algumas estruturas são declaradas:

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host = socket.gethostname()

port = 7777

try:
    s.bind((host, port))
except socket.error as msg:
    print("bind failed" + str(msg))
    sys.exit()


s.listen(10)

process_sockets_list = []
process_list = []
neighbor_list = []
msg_token = ""
```

Estruturas como a lista de sockets de processos, a lista de processos, de vizinhos e o token de mensagem recebido. Agora falando sobre o cliente:

```python
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

host = socket.gethostname()

to_port = 7777
s.connect((host, to_port))

my_id = "0"

s.send(my_id.encode('utf-8'))

leader="-1"


def initiate_election(s):

    time.sleep(1)

    s.send(my_id.encode('utf-8'))

    print("Token enviado: " + my_id)
    print("Eleição iniciada")
```

```python
def Ring_Election_Algorithm(s):

    while True:
        global leader
        try:

            s.settimeout(15)
            received = s.recv(1024)
            s.settimeout(None)

            received_token_list = received.decode('utf-8')

        except socket.timeout:
            leader = "0"
            initiate_election(s)
            continue


        if my_id in received_token_list and "Coordenador: " not in received_token_list and "hello" not in received_token_list:

            leader = max(received_token_list)
            forwarding_leader = "Coordenador: " + leader

            time.sleep(1)

            s.send(forwarding_leader.encode('utf-8'))

        elif my_id not in received_token_list and "Coordenador: " not in received_token_list and "hello" not in received_token_list :

            print("Token recebida: " + received_token_list)

            leader = "0"

            received_token_list = received_token_list + " " + my_id

            time.sleep(1)

            s.send(received_token_list.encode('utf-8'))
            print("Adicionando token: " + received_token_list)
```

```python
        elif ("hello" in received_token_list or "Coordenador: " in received_token_list )and leader=="-1"  :
                leader="0"
                initiate_election(s)


        elif "Coordenador: " in received_token_list and leader not in received_token_list :
            print(received_token_list)

            le=received_token_list.split()
            leader=le[1]

            time.sleep(1)

            s.send(received_token_list.encode('utf-8'))


        else :
            if leader=="-1" or leader=="0":
                continue
            else :

                print(received_token_list)

                communicate = "hello" + " from " + my_id

                time.sleep(1)

                s.send(communicate.encode('utf-8'))
                continue

recv_thread = threading.Thread(target=Ring_Election_Algorithm, args=(s,))
recv_thread.start()
recv_thread.join()
s.close()
```

Como pode ser observado nas imagens acima, inicialmente o cliente conecta no servidor, e fica esperando por comunicações, podendo até iniciar eleições em certos casos. As eleições são feitas por passagem de mensagens, e é sempre verificado qual o coordenador do anel. Lembrando novamente, que se um coordenador sair, outra eleição é feita e um novo coordenador será eleito.

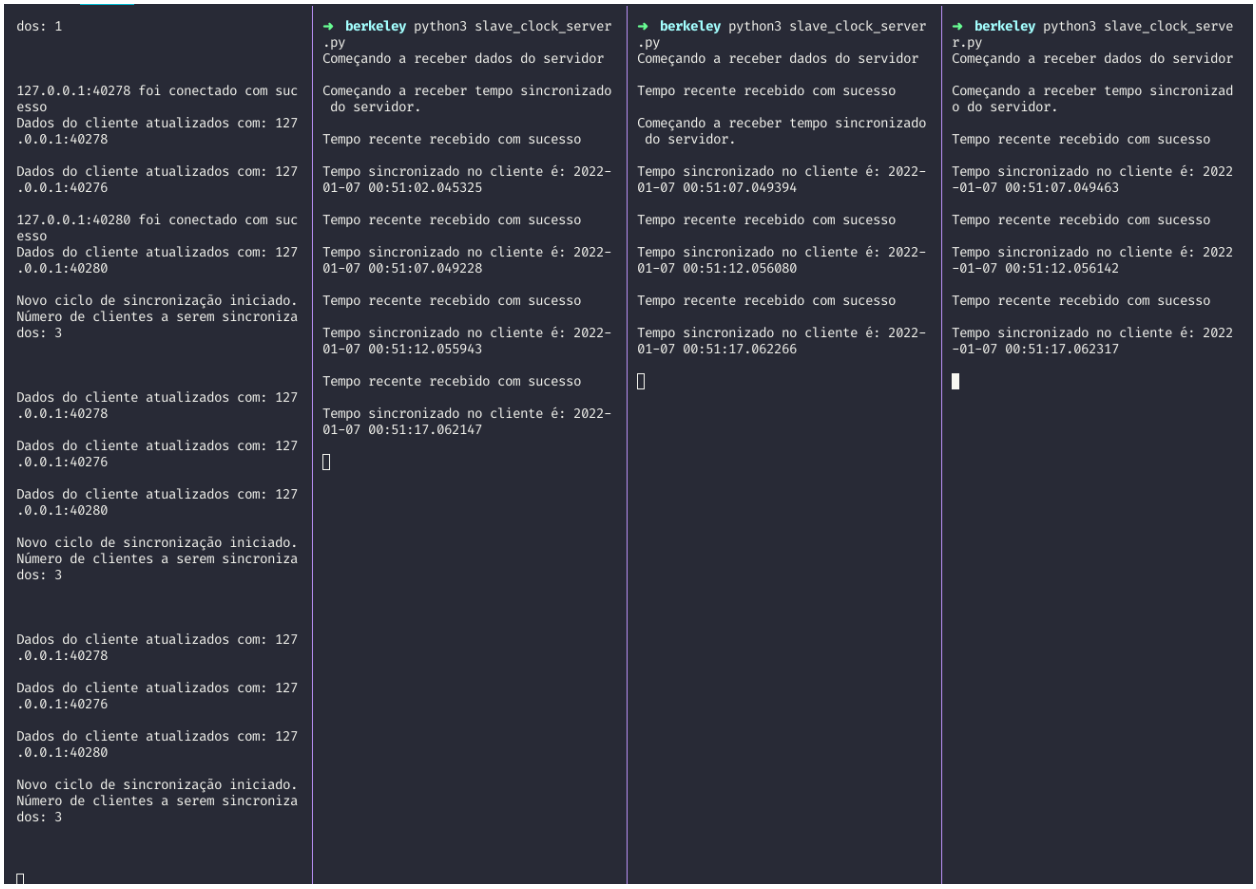
Dessa forma, é modelado o algoritmo de eleição do anel. A seguir, serão listados os testes executados.

Testes executados

Para executar os softwares, é essencial que se tenha o python versão 3.6 ou superior instalado. Além disso, é necessário ter o gerenciador de pacotes pip e instalar o pacote python-dateutil (pip install python-dateutil).

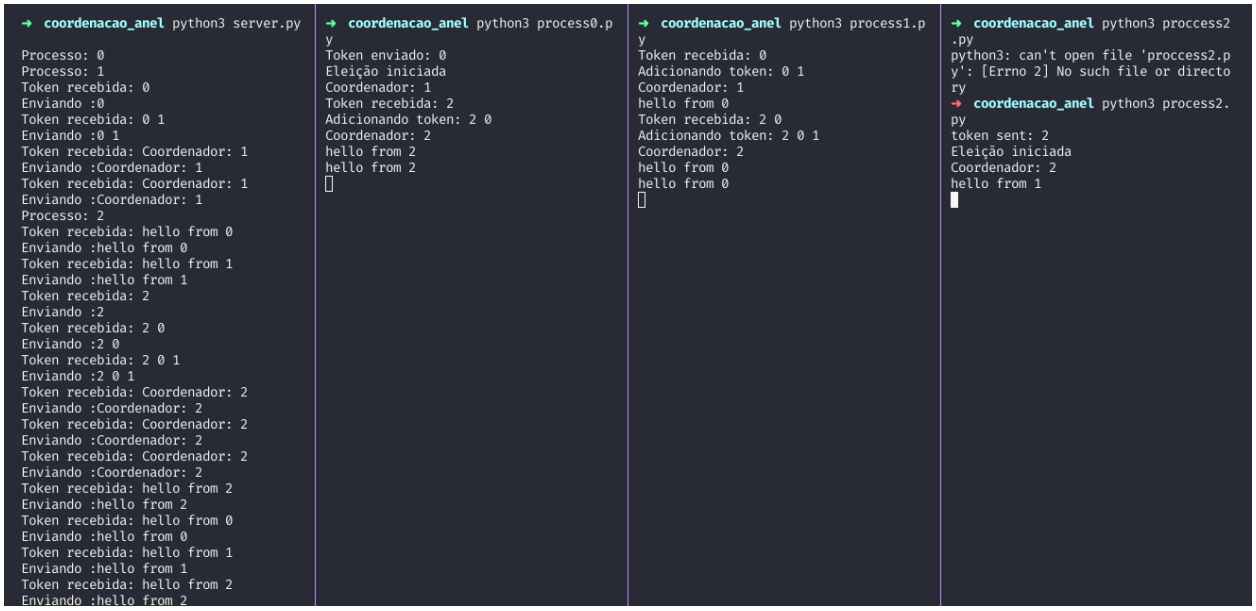
Algoritmo de Berkeley

Para rodar o algoritmo de berkeley, basta rodar 1 instância do servidor master (pasta berkeley, arquivo master_clock_server.py) e 3 ou mais do servidor slave (pasta berkeley, arquivo slave_clock_server.py), que ele começará a sincronizar os relógios e dar os resultados após alguns segundos:



Algoritmo de eleição do anel

Muito simples de executar também, entre na pasta coordenacao_anel e rode todos os arquivos, começando pelo server.py e indo de process0 para proccess3. Ao fazer isso, você poderá verificar que as eleições já começam a ser feitas:



Perceba inclusive que o coordenador é trocado nesse exemplo, demonstrando que o algoritmo funciona corretamente.

Conclusão

Ambos os algoritmos são algoritmos muito efetivos, que resolvem grandes problemas de sistemas distribuídos, e muito conhecimento foi adquirido ao implementar ambos. As maiores dificuldades se encontraram em colocar tudo em rede, e também entender o escopo do trabalho e as tarefas específicas a serem feitas. No geral, foi um ótimo trabalho que agregou muito aprendizado.