UNIVERSIDADE FEDERAL DO ABC

Sistemas Distribuídos

Projeto de Programação – KV Store

Pedro Victor Marcelino Jordão Motta RA: 11201921599

Santo André
2023

Screencast

Segue o link do screencast disponível no youtube e Google Drive:

- Youtube: https://youtu.be/ZVRIyLs2VIE
- Google Drive: https://drive.google.com/file/d/1j57oAu4XwqkLIzvfHUbip8Xdrn7F8oXO/view?usp=sharing

Projeto

Seguindo as instruções propostas no documento, o sistema foi dividido em um server.py e cliente.py e message.py .Em ambos foi utilizado estruturas de JSON para lidar com informações pela facilidade de manipulação e por ser uma estrutura amplamente utilizada. A explicação em alto nível das funções implementadas em cada programa será feita a seguir, além da explicação e exemplos das estruturas utilizadas:

message.py

- Criação da classe Message a função init inicializa todos os parâmetros.

```
1   class Message:
2
3       """
4           Uma classe que representa um objeto de mensagem.
5
6           Atributos:
7               method (str): O método da mensagem.
8               value (str): O valor da mensagem.
9               key (str, opcional): A chave associada à mensagem (padrão é None).
10              timestamp (str, opcional): O timestamp da mensagem (padrão é None).
11      """
12
13      def __init__(self, method, value, key=None, timestamp=None):
14          """
15          Inicializa um objeto da classe Message.
16          """
17          self.method = method
18          self.value = value
19          self.key=key
20          self.timestamp= timestamp
```

Imagem 1.

- A função __str___(linha 22) retorna uma representação em string do objeto, a função to_json (linha 29) converte o objeto em um json, a função from_json (linha 40) cria um objeto a partir de um objeto Messagem, o decorador @classmethod é responsável pode possibilitar que o método seja chamado na classe, em vez de uma instância da classe.

```python
    def __str__(self):
        """
            Retorna uma representação em string do objeto Message.
        """
        return f"Message: method={self.method}, value={self.value}, key={self.key}, timestamp={self.timestamp}"


    def to_json(self):
        """
        Converte o objeto Message em um objeto JSON.
        """
        return {
            "method": self.method,
            "value": self.value,
            "key": self.key,
            "timestamp": self.timestamp,
        }

    @classmethod
    def from_json(cls, json_data):
        """
        Cria um objeto Message a partir de um objeto JSON.

        O decorador @classmethod em Python diz ao Python que o método from_json()
        é um método de classe. Isso significa que o método pode ser chamado na classe em si,
        em vez de em uma instância da classe.
        """
        return cls(json_data["method"], json_data["value"],json_data["key"],json_data["timestamp"])
```

Imagem 2.

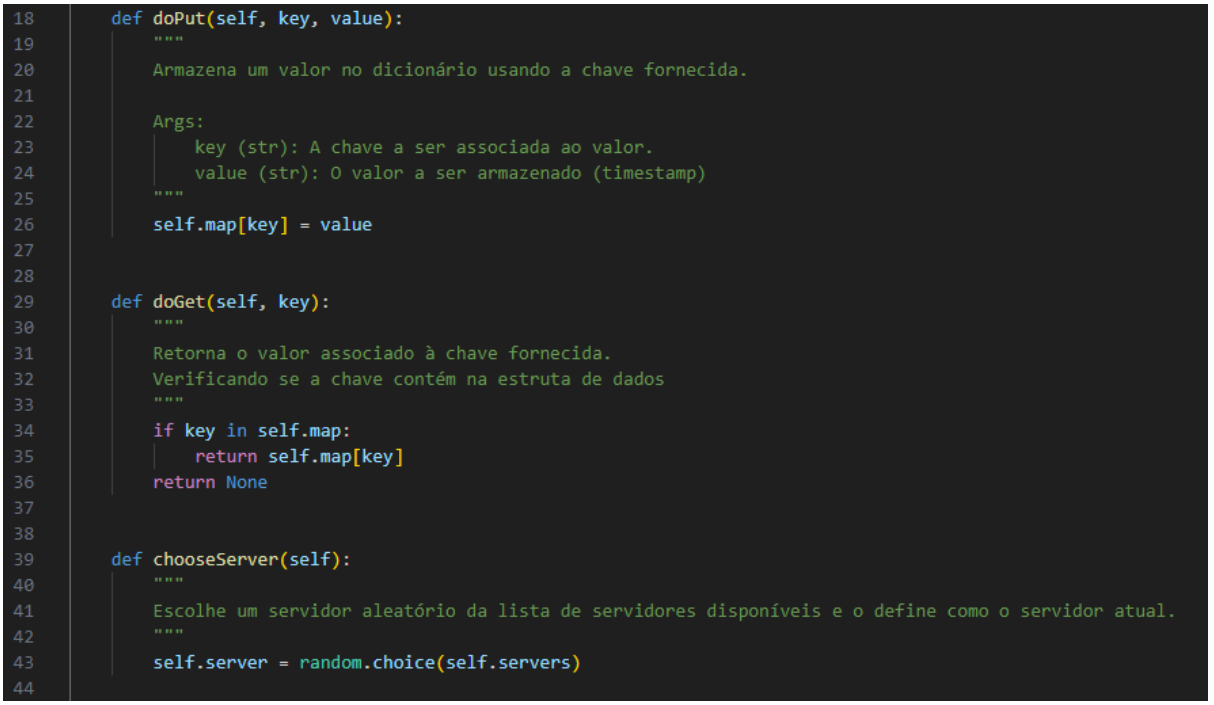
cliente.py

- Criação da classe cliente e a função init inicializa todos os parâmetros.

```python
class Client:
    def __init__(self):
        """
        Inicializa um objeto da classe Client.
        """
        self.servers = [('127.0.0.1', 10097), ('127.0.0.1', 10098), ('127.0.0.1', 10099)]
        self.server = None
        self.map = {}
```

Imagem 3.

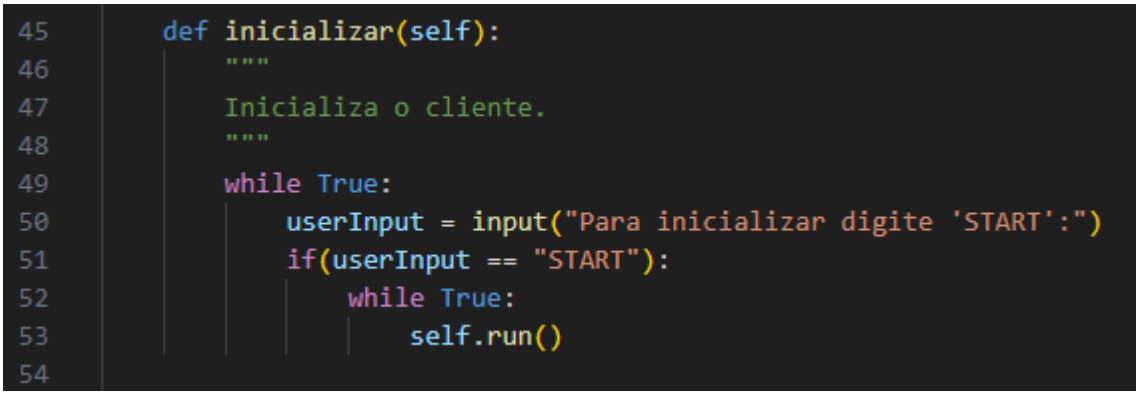
- A função doPut (linha 18) é responsável armazenar o valor em um dicionário usando a chave fornecida, o cliente sempre irá guardar a chave e o valor do timestamp associado. A função doGet (linha 29) retorna um valor associado à chave pedida. Já a função chooseServer (linha 39) escolhe um servidor de forma aleatório para o cliente se conectar.



```
18    def doPut(self, key, value):
19        """
20        Armazena um valor no dicionário usando a chave fornecida.
21
22        Args:
23            key (str): A chave a ser associada ao valor.
24            value (str): O valor a ser armazenado (timestamp)
25        """
26        self.map[key] = value
27
28
29    def doGet(self, key):
30        """
31        Retorna o valor associado à chave fornecida.
32        Verificando se a chave contém na estruta de dados
33        """
34        if key in self.map:
35            return self.map[key]
36        return None
37
38
39    def chooseServer(self):
40        """
41        Escolhe um servidor aleatório da lista de servidores disponíveis e o define como o servidor atual.
42        """
43        self.server = random.choice(self.servers)
44
```

Imagem 4.

- A função inicializar (linha 45) é responsável por inicialização do menu do cliente e chamar o método .run() que será explicado logo abaixo.



```
45        def inicializar(self):
46            """
47            Inicializa o cliente.
48            """
49            while True:
50                userInput = input("Para inicializar digite 'START':")
51                if(userInput == "START"):
52                    while True:
53                        self.run()
54
```

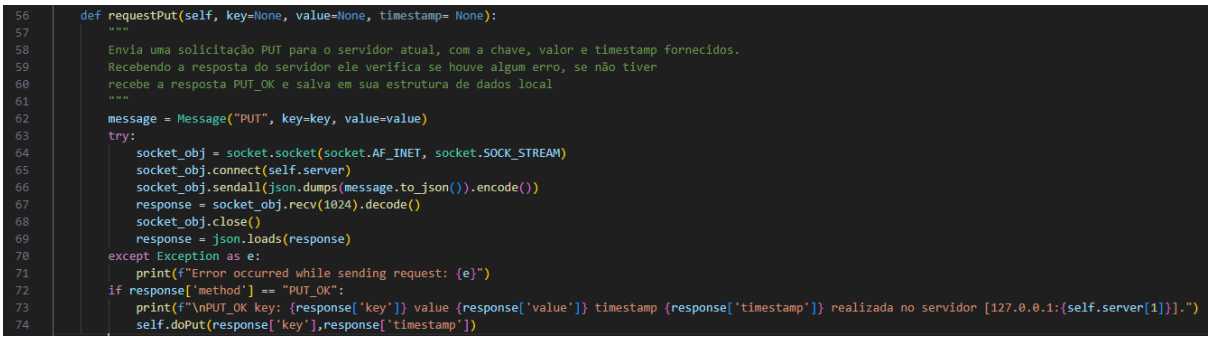
Imagem 5.

- A função run(linha 108) é responsável por ter o menu interativo para o cliente, com as opções de PUT e GET e fazer a tratativa do request, é válido ressaltar que a linhas 122 e 125 usam threads para receber a resposta do servidor para o cliente não ficar travado, assim ele pode realizar diversos puts e diversos get ao mesmo tempo, sem esperar a resposta do server (que pode chegar com atraso). Válido ressaltar que o cliente sempre irá se conectar com um server diferente a cada requisição (linha 113)

```
107
108     def run(self):
109         """
110         Executa o cliente.
111         """
112         while True:
113             self.chooseServer()
114             print("Opções:")
115             print("1. PUT")
116             print("2. GET")
117             acao = input("Escolha a ação (1-3): ")
118
119             if acao == "1":
120                 key = input("Digite a chave(Key):")
121                 value = input("Digite o valor(Value):")
122                 threading.Thread(target=self.requestPut, args=(key , value)).start()
123             elif acao == "2":
124                 key = input("Digite a chave(Key):")
125                 threading.Thread(target=self.requestGET, args=(key)).start()
126             else:
127                 print("Opção Inválida")
128
129
130
131 client = Client()
132 client.inicializar()
```

Imagem 6.

- A função requestPut (linha 56), é responsável por enviar PUT para o servidor atual, com a chave e valor e timestamp fornecido, que será None. Essa requisição é tratado em uma thread e fecha a conexão, após receber a resposta, caso ocorra tudo certo a estrutura de dados local do cliente é atualizado



```
56  def requestPut(self, key=None, value=None, timestamp= None):
57      """
58      Envia uma solicitação PUT para o servidor atual, com a chave, valor e timestamp fornecidos.
59      Recebendo a resposta do servidor ele verifica se houve algum erro, se não tiver
60      recebe a resposta PUT_OK e salva em sua estrutura de dados local
61      """
62      message = Message("PUT", key=key, value=value)
63      try:
64          socket_obj = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
65          socket_obj.connect(self.server)
66          socket_obj.sendall(json.dumps(message.to_json()).encode())
67          response = socket_obj.recv(1024).decode()
68          socket_obj.close()
69          response = json.loads(response)
70      except Exception as e:
71          print(f"Error occurred while sending request: {e}")
72      if response['method'] == "PUT_OK":
73          print(f"\nPUT_OK key: {response['key']} value {response['value']} timestamp {response['timestamp']} realizada no servidor [127.0.0.1:{self.server[1]}].")
74          self.doPut(response['key'],response['timestamp'])
```

Imagem 7.

- A função requestGet (linha 77), é responsável por enviar Get para o servidor atual, com a chave e valor e timestamp local, caso o cliente não tenha um timestamp associado aquela chave irá enviar None. Essa requisição é tratada em uma thread e fecha a conexão, após receber a resposta, caso ocorra tudo certo a estrutura de dados local do cliente é atualizada.

Entretanto, existe a opção que a chave não existir no servidor ou o servidor pedir para tentar novamente para outro server ou mais tarde

```python
def requestGET(self, key):
    """
    Envia uma solicitação GET para o servidor atual, com a chave fornecida.
    Caso não tenha a cahve em sua estruta, define o localTimestamp como None
    Nesse caso, pode receber três respostas do server:
    Primeira: O server não tem a chave
    Segunda: TRY_OTHER_SERVER_OR_LATER, ou seja, devemos fazer a solicitação posterior
    Terceira: GET_OK, e atualiza o valor da estrutura de dados local
    """
    timer = self.doGet(key)
    localTimestamp = None if timer == None else timer
    message = Message("GET", value= timer , key= key)
    try:
        socket_obj = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        socket_obj.connect(self.server)
        socket_obj.sendall(json.dumps(message.to_json()).encode())
        response = socket_obj.recv(1024).decode()
        socket_obj.close()
        response = json.loads(response)
    except Exception as e:
        print(f"Error occurred while sending request: {e}")

    if response['method'] == "NULL":
        print("\nKEY NOT FOUND")
    elif response['method'] == "TRY_OTHER_SERVER_OR_LATER":
        print("\nTRY_OTHER_SERVER_OR_LATER")
    else:
        self.doPut(key,response['value'][1])
        print(f"\nGET key: {key} value: {response['value'][0]} obtido do servidor [127.0.0.1:{self.server[1]}], meu timestamp {localTimestamp} e do servidor {response['value'][1]}")
```

Imagem 8.

server.py

- Criação da classe cliente e a função init inicializa todos os parâmetros.

```python
class Server:

    def __init__(self):
        """
        Inicializa um objeto da classe Server.
        """
        self.ip = "127.0.0.1"
        self.port = None
        self.leaderIp = self.ip
        self.portaLider = None
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.portasValidas = [10097, 10098, 10099]
        self.isLider = None
        self.map = {}
```

Imagem 9.

- Função put (linha 22), é responsável por armazenar em um dicionário usando a chave fornecida, mas diferente do cliente, ele vai armazenar uma tupla. A função get (linha 32) retorna a tupla associado a chave fornecida.

```python
    def put(self, key, value):
        """
        Armazena um valor no dicionário usando a chave fornecida.

        Args:
            key (str): A chave a ser associada ao valor.
            value (str): O valor a ser armazenado, o valor é um tupla (value,timestamp)
        """
        self.map[key] = value

    def get(self, key):
        """
        Retorna o valor associado à chave fornecida.
        Returns:
            str or None: O valor associado à chave, ou None se a chave não existir no dicionário.
        """
        if key in self.map:
            return self.map[key]
        return None
```

Imagem 10.

- Função setPortSettings (linha 42), é responsável configurar as portas do server e verifica a disponibilidade dos servidores e obtém a porta do líder. Além de chamar a função setPort e setportaLider

```python
    def setPortSettings(self):
        """
        Configura as definições de porta do servidor.
        Verifica a disponibilidade dos servidores e obtém a porta do líder do cluster.
        """
        portaLider = None
        activePorts = []
        for port in self.portasValidas:
            with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
                try:
                    sock.connect((self.ip, port))
                    sock.sendall(json.dumps(Message("isLider", None).to_json()).encode())
                    isLider = sock.recv(1024).decode()
                    message = Message.from_json(json.loads(isLider))
                    if(message.value == True):
                        portaLider = port
                        activePorts.append(port)
                    else:
                        activePorts.append(port)
                except Exception as e:
                    pass
        self.setPort(activePorts)
        self.setportaLider(portaLider)
```

Imagem 11.

- Função setportaLider (linha 66), é responsável definir a porta do líder e faz verificações sobre as portas válidas do líder.

```
66      def setportaLider(self, portaLider):
67          """
68          Define a porta do servidor líder do cluster. E faz verificações
69          sobre portas validas do lider
70          """
71          while True:
72              try:
73                  inputPort = int(input("Digite o número da porta do servidor Líder: "))
74                  if inputPort in self.portasValidas:
75                      if portaLider == None:
76                          if inputPort == self.port:
77                              self.isLider = True
78                              break
79                      else:
80                          if inputPort == portaLider:
81                              self.isLider = False
82                              self.portaLider = inputPort
83                              break
84                  else:
85                      print("Porta inválida ou já utilizada")
86              except ValueError:
87                  pass
```

Imagem 12.

- Função setPort (linha 89), é responsável por definir a porta do servidor e faz verificações para ver se a porta já está sendo utilizada e se ela é válida.

```
89      def setPort(self, activePorts):
90          """
91          Define a porta do servidor e faz verificação para ver a porta já
92          está sendo utilizada e se ela é valida
93          """
94          while True:
95              try:
96                  inputPort = int(input("Digite a porta do servidor: "))
97                  if inputPort in self.portasValidas:
98                      if not (inputPort in activePorts):
99                          self.port = inputPort
100                         break
101                 else:
102                     print("Porta inválida ou já utilizada")
103             except ValueError:
104                 pass
```

Imagem 13.

- Função start (linha 106), é responsável inicializar o servidor e utilizar os threads para lidar com diversas requisições definir a porta do servidor e faz verificações para ver se a porta já está sendo utilizada e se ela é válida.

```
106     def start(self):
107         """
108         Inicia o servidor. Utiliza threads  para lidar com diversas requisições
109         """
110         self.setPortSettings()
111 
112         self.socket.bind((self.ip, self.port))
113         self.socket.listen()
114         print(f"Servidor escutando {self.ip}:{self.port}")
115         while True:
116             conn, addr = self.socket.accept()
117             threading.Thread(target=self.handleClients, args=(conn, addr)).start()
```

Imagem 14.

- A Função handleClients (linha 119), é responsável por lidar com as solicitações de clientes e de outros servidores e fecha a conexão para não consumir recursos.

```
119     def handleClients(self, conn, addr):
120         """
121         Lida com as solicitações dos clientes. Dependendo de cada método
122         Método IsLider, manda menssagem se ou não lider
123         Método PUT, manda a mensagem para a função doPut
124         Método Get, manda a mensagem para a função doGet
125         Método REPLICATION, ativa o doReplication que fará a replicação dos dados
126         E posteriormente fecha a conexão para não consumir recursos
127         """
128         data = conn.recv(1024).decode()
129         message = Message.from_json(json.loads(data))
130 
131         if message.method == "isLider":
132             response = self.isLider
133             response_message = Message("isLiderResponse", response)
134             conn.sendall(json.dumps(response_message.to_json()).encode())
135 
136         elif message.method == "PUT":
137             self.doPut(conn, message)
138 
139         elif message.method == "GET":
140             self.doGet(conn, message)
141         elif message.method == "REPLICATION":
142             self.doReplication(conn, message)
143 
144         conn.close()
```

Imagem 15.

- A Função doReplication (linha 147), é responsável por receber uma requisição replication e atualizar sua estrutura de dados local, e retornar ao servidor um Replication_OK.

```python
147     def doReplication(self, conn, message):
148         """
149         Realiza a replicação de dados, após receber uma requisição REPLICATION
150         e envia de volta REPLICATION_OK, quando armazenar em sua estrutura local
151         """
152         valor = message.value[0]
153         timestamp= message.value[1]
154         chave= message.key
155 
156         print(f"REPLICATION key:{chave} value:{valor} ts:{timestamp}.")
157 
158         if chave in self.map:
159             self.map[chave] = (valor, timestamp)
160         else:
161             self.put(chave, (valor, timestamp))
162 
163 
164         conn.sendall(json.dumps(Message("REPLICATION_OK", valor, chave ,timestamp).to_json()).encode())
```

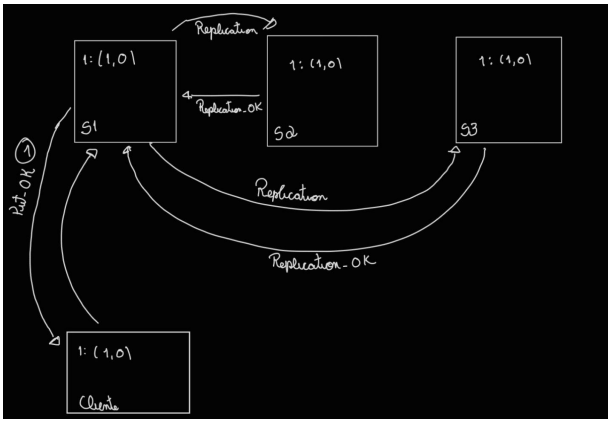
Imagem 16.

- Função doGet (linha 166), é responsável processar uma solicitação get. No primeiro momento ele verifica se a chave existe e faz um get em sua estrutura de dados local. Caso o cliente enviou uma mensagem com timestamp vazio e existe essa chave no servidor, ele atribui seu próprio timestamp para devolver ao cliente.
  Caso não exista essa chave ele irá retornar um erro, visto a partir da linha 179. Caso a o timestamp do cliente seja maior que a do servidor, ele irá retornar o erro TRY_OTHER_SERVER_OR_LATERM, visto a partir da linha 182. Caso, ao contrário, ele irá retornar ao cliente o valor daquela chave.

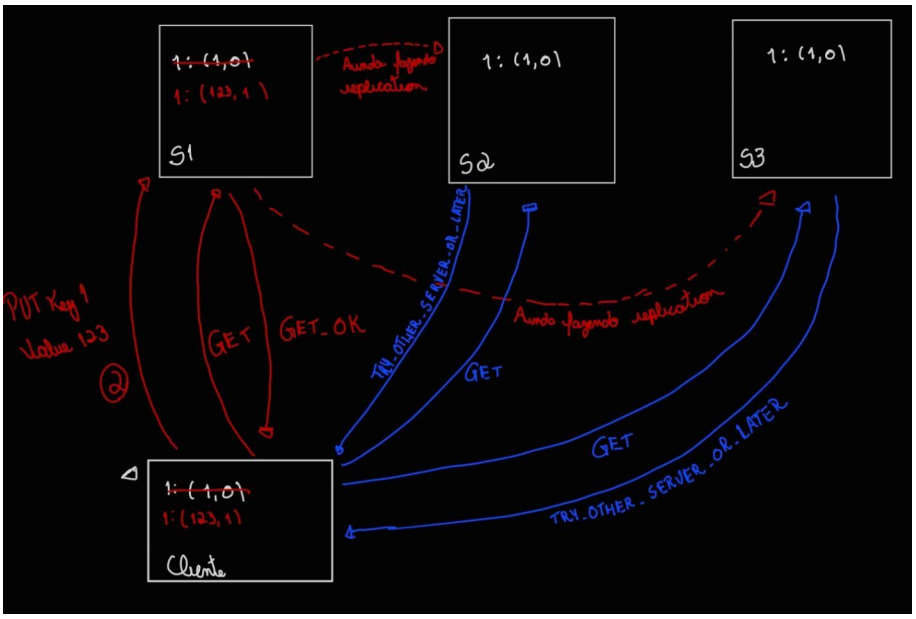
```python
166     def doGet(self, conn, message):
167         """
168         Processa uma solicitação GET.
169         -Caso o cliente não tenha a chave em sua estrutura local e pedir um get dela, ele recebe o timestamp armazenado no servidor, visto na linha (173)
170         - Caso não exista essa chave em sua estrutura, o servidor envia NULL ao cliente
171         - Caso o cliente tenha um ts maior que do server, ele envia TRY_OTHER_SERVER_OR_LATER
172         - Caso esteja tudo correto, envia GET_OK
173         """
174         item = self.get(message.key)
175 
176         if(message.value==None and item!=None):
177             message.value=item[1]
178 
179         if item == None:
180             print(f"Cliente {conn.getpeername()[0]}:{conn.getpeername()[1]} GET key:{message.key} ts:None, Meu ts é None, portanto devolvendo NULL")
181             conn.sendall(json.dumps(Message("NULL", item, None).to_json()).encode())
182         elif item[1] is not None and item[1] < message.value:
183             print(f"Cliente {conn.getpeername()[0]}:{conn.getpeername()[1]} GET key:{message.key} ts:{message.value}, Meu ts é {item[1]}, portanto devolvendo TRY_OTHER_SERVER_OR_LATER")
184             conn.sendall(json.dumps(Message("TRY_OTHER_SERVER_OR_LATER", None).to_json()).encode())
185         else:
186             print(f"Cliente {conn.getpeername()[0]}:{conn.getpeername()[1]} GET key:{message.key} ts:{message.value}, Meu ts é {item[1]}, portanto devolvendo GET_OK")
187             conn.sendall(json.dumps(Message("GET_OK", item).to_json()).encode())
188 
```

Imagem 17.

O erro TRY_OTHER_SERVER_OR_LATER, será simulado da seguinte maneira e necessitando apenas de um cliente, o primeiro passo do cliente realizar um put normal e receber um put_ok:



O segundo passo, é realizar um novo put na mesma chave e realizar um get antes de receber um put_ok, assim temos um novo valor de timestamp, e realizando um get para os servidores que tem atraso para recebermos o erro



- Função doServerPut (linha 197), é acionado quando é realizado um Put em um servidor que não é líder, com tal ocorrência o servidor irá encaminhar a requisição ao líder e posteriormente, irá mandar a mensagem ao cliente, caso ocorra tudo certo.

```
197     def doServerPut(self, conn, message):
198         """
199         Um servidor que não é lider processa uma solicitação PUT, mandando a
200         menssagem para o lider e pedindo para ele recplicar, ele recebe uma menssagem
201         put_ok do lider, e encaminha esse put_ok para o cliente
202         """
203         with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
204                     try:
205                         sock.connect((self.ip, self.portaLider))
206                         sock.sendall(json.dumps(message.to_json()).encode())
207                         replication = sock.recv(1024).decode()
208                         response = Message.from_json(json.loads(replication))
209 
210                     except Exception as e:
211                         pass
212 
213         valor = response.value
214         chave = response.key
215         timestamp= response.timestamp
216         conn.sendall(json.dumps(Message("PUT_OK",valor, chave ,timestamp).to_json()).encode())
217 
```

Imagem 18.

- Função doLeaderPut (linha 219), é o processo que o líder realiza, quando recebe uma requisição put ou quando é encaminhado uma mensagem put. Ele manda para o servers uma mensagem de REPLICATION, e atrasa em 10 segundos o encaminhamento dessa mensagem (simular um atraso de rede,

por exemplo). Quando ele receber três respostas de REPLICATION_OK (uma dele mesmo e mais duas) ele envia ao cliente put_OK

```python
    def doLeaderPut(self, conn, message):
        """
        O servidor que é lider processa uma solicitação PUT, mandando a
        menssagem REPLICATION para ele mesmo e para outros server, após ele
        receber a quantidade de mensagem de Replication_OK ele recebe uma menssagem
        put_ok
        """
        timestamp = time.time()
        valor = message.value
        chave= message.key


        serveResponse = []
        if chave in self.map:
            self.map[chave] = (valor, timestamp)
        else:
            self.put(chave, (valor, timestamp))

        message.method = "REPLICATION"
        message.value = (valor, timestamp)
        message.timestamp=timestamp
        for port in self.portasValidas:
            if port != self.portaLider:
                time.sleep(10)
                with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
                    try:
                        sock.connect((self.ip, port))
                        sock.sendall(json.dumps(message.to_json()).encode())
                        serveResponse.append( sock.recv(1024).decode())

                    except Exception as e:
                        pass

        if(len(serveResponse)==3):
            print(f"Enviando PUT_OK ao Cliente {conn.getpeername()[0]}:{conn.getpeername()[1]} da key:{chave} ts:{timestamp}")
            conn.sendall(json.dumps(Message("PUT_OK",valor, chave ,timestamp).to_json()).encode())

        
Server().start()
```

Imagem 19.

Threads

Foi utilizado Threads no cliente.py:

- Na linha 122 e 125, dentro da função run(), em ambos os casos, permite que o cliente realize diversas requisições aos servidores de forma simultânea, não necessitando esperar uma resposta do servidor para realizar uma nova ação.

Foi utilizado Threads no server.py:

- Na linha 117 para que cada requisição seja tratada em uma thread separada, permitindo que receba diversas requisições ao mesmo tempo.

Código

Google Drive:

https://drive.google.com/file/d/1jbY0rukBcl8iMYSxygZgQd3ON3amu13v/view?usp=sharing

GitsGitHub:
https://gist.github.com/pedrovmjm/b8513ed3503ce33a3359e07844c8ee97