Especificação do Projeto: Sistema Distribuído de Processamento de Documentos com Ordenação Total de Eventos

Este documento especifica um sistema distribuído para processamento de documentos, utilizando containers Docker e relógios vetoriais para garantir a **ordenação total** de eventos no log. O sistema será composto por uma aplicação principal e vários microsserviços.

1. Requisitos Funcionais:

- **Upload de Arquivos:** A aplicação principal permitirá o upload de arquivos PDF e imagens (PNG, JPG, TIFF).
- **Conversão PDF para TXT:** Um microsserviço converterá arquivos PDF para texto plano (TXT).
- **Redução de Resolução de PDF:** Um microsserviço reduzirá a resolução de arquivos PDF.
- **Redimensionamento de Imagens:** Um microsserviço redimensionará imagens.
- **Conversão de Formatos de Imagens:** Um microsserviço converterá imagens entre os formatos PNG, JPG e TIFF.
- **Log Centralizado com Ordenação Total:** Um serviço de log centralizado registrará todos os eventos, garantindo a **ordenação total** dos eventos utilizando timestamps e relógios vetoriais para resolução de conflitos de tempo.
- **Aplicação Principal:** Uma aplicação principal coordenará as operações, recebendo requisições do usuário, encaminhando-as aos microsserviços apropriados e retornando os resultados. Esta aplicação terá endpoints para cada funcionalidade.

2. Requisitos Não Funcionais:

- **Escalabilidade:** O sistema deverá ser escalável, permitindo a adição de novos microsserviços e a distribuição de carga.
- **Consistência:** A ordem dos eventos no log deve ser consistente e corresponder à ordem de ocorrência real.
- **Tolerância a Falhas:** O sistema deverá ser tolerante a falhas em microsserviços individuais. A falha de um microsserviço não deve impedir o funcionamento dos demais.
- **Segurança:** A comunicação entre os microsserviços e a aplicação principal deverá ser segura. (Este requisito pode ser simplificado para o escopo deste exemplo)
- **Tecnologia:** Utilizar Python, Flask, Docker, e relógios vetoriais.

3. Arquitetura:

O sistema será baseado em microsserviços, cada um executando em um container Docker separado. A comunicação entre os microsserviços e a aplicação principal será realizada via requisições HTTP. O log centralizado utilizará um mecanismo baseado em timestamps e relógios vetoriais para garantir a ordenação total dos eventos, resolvendo conflitos de tempo.

4. Detalhes de Implementação (Relógios Vetoriais e Ordenação Total):

Cada microsserviço manterá seu próprio relógio vetorial e registrará um timestamp para cada evento. Ao processar uma requisição, o microsserviço atualizará seu relógio, registrará o

timestamp e enviará a mensagem de log para o servidor central, juntamente com o timestamp e uma cópia do relógio. O servidor central utilizará os timestamps e os relógios vetoriais para ordenar as mensagens de log:

- **Primaria:** Ordenação por timestamp.
- **Secundária:** Em caso de timestamps iguais (ou muito próximos, levando em conta a precisão do relógio), será usada a comparação lexicográfica dos relógios vetoriais para desempatar.

Solução Completa

Estrutura de Pastas: (sugestão)

```
projeto_lab_distribuido/

├── central_log/

|   ├── log_service.py

|   └── Dockerfile

├── pdf2txt/

|   ├── app.py

|   ├── requirements.txt

|   └── Dockerfile

├── reduce_resolution/

|   ├── app.py

|   ├── requirements.txt

|   └── Dockerfile

├── resize_images/

|   ├── app.py

|   ├── requirements.txt

|   └── Dockerfile

├── convert_images/

|   ├── app.py

|   ├── requirements.txt

|   └── Dockerfile

└── vector_clock.py # Módulo para o relógio vetorial (reutilizável)
```

1. Módulo vector_clock.py: (exemplo)

```
import copy

class VectorClock:
    def __init__(self, num_processes, process_id):
        self.clock = [0] * num_processes
        self.process_id = process_id

    def tick(self):
        self.clock[self.process_id] += 1

    def update(self, other_clock):
        for i in range(len(self.clock)):
            self.clock[i] = max(self.clock[i], other_clock[i])

    def get_clock(self):
        return copy.deepcopy(self.clock)

    def __lt__(self, other):
        for i in range(len(self.clock)):
            if self.clock[i] > other.clock[i]:
                return False
        return True

    def __eq__(self, other):
        return self.clock == other.clock

    def __le__(self,other):
        return self < other or self == other

    def __gt__(self,other):
        return not self <= other

    def __ge__(self,other):
        return not self < other

    def __ne__(self, other):
        return not self == other

    def __repr__(self):
        return str(self.clock)
```

2. Serviço de Log Central (central_log/log_service.py):

```
import socket
import pickle
import threading
import time
from vector_clock import VectorClock
```

```
class LogService:
    def __init__(self, port):
        self.port = port
        self.logs = []
        self.lock = threading.Lock()

    def start(self):
        threading.Thread(target=self.receive_log_messages).start()

    def receive_log_messages(self):
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.bind(('', self.port))
        sock.listen(1)
        print(f"Serviço de log central iniciado na porta {self.port}")
        while True:
            conn, addr = sock.accept()
            with conn:
                print(f'Conectado por {addr}')
                while True:
                    data = conn.recv(1024)
                    if not data:
                        break
                    self.process_log_message(data)
        sock.close()

    def process_log_message(self, data):
        message, vector_clock, timestamp = pickle.loads(data) # recebe timestamp
        with self.lock:
            self.logs.append((timestamp, vector_clock, message)) #armazena com timestamp
            self.logs.sort() # Ordena primeiro por timestamp, depois por relógio
vetorial.
            print(f"Log registrado (ordenado): {self.logs}")

if __name__ == "__main__":
    log_service = LogService(port=6000)
    log_service.start()
```

3. Aplicação Principal (main_app/app.py):

```
from flask import Flask, request, jsonify
import requests
import json
import os
from vector_clock import VectorClock
import time
import socket
import pickle

app = Flask(__name__)
num_processes = 6
```

```
process_id = 5
my_clock = VectorClock(num_processes, process_id)

SERVICES = {
    "pdf2txt": {"url": "http://pdf2txt:5001/", "vector_clock": VectorClock(num_processes,
0)},
    "reduce_resolution": {"url": "http://reduce_resolution:5002/", "vector_clock":
VectorClock(num_processes, 1)},
    "resize_images": {"url": "http://resize_images:5003/", "vector_clock":
VectorClock(num_processes, 2)},
    "convert_images": {"url": "http://convert_images:5004/", "vector_clock":
VectorClock(num_processes, 3)}
}

def log_event(message):
    HOST = 'localhost'
    PORT = 6000
    my_clock.tick()
    timestamp = time.time()
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect((HOST, PORT))
        s.sendall(pickle.dumps((message, my_clock.get_clock(), timestamp)))


@app.route('/pdf2txt', methods=['POST'])
def pdf2txt_route():
    try:
        file = request.files['file']
        files = {'file': file}
        response = requests.post(SERVICES["pdf2txt"]["url"], files=files)
        log_event(f"Requisição pdf2txt: {response.status_code}")
        return response.content, response.status_code
    except Exception as e:
        log_event(f"Erro em pdf2txt: {e}")
        return jsonify({"error": str(e)}), 500

# ... (endpoints semelhantes para os outros serviços - Implemente os outros endpoints
aqui)

@app.route('/healthcheck')
def health_check():
    return 'ok'

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0', port=5005)
```

4. Serviço Web (Exemplo: pdf2txt/app.py):

```
from flask import Flask, request, send_from_directory
import os
import subprocess
import json
```

```
from vector_clock import VectorClock
import socket
import pickle
import time

app = Flask(__name__)
UPLOAD_FOLDER = 'uploads'
app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER
os.makedirs(UPLOAD_FOLDER, exist_ok=True)
num_processes = 6
process_id = 0
my_clock = VectorClock(num_processes, process_id)

def log_event(message, log_server_address=('localhost', 6000)):
    my_clock.tick()
    timestamp = time.time()
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect(log_server_address)
        s.sendall(pickle.dumps((message, my_clock.get_clock(), timestamp)))

@app.route('/', methods=['POST'])
def upload_file():
    if request.method == 'POST':
        file = request.files['file']
        if file.filename != '':
            file_path = os.path.join(app.config['UPLOAD_FOLDER'], file.filename)
            file.save(file_path)
            output_filename = file_path.replace(".pdf", ".txt")
            try:
                subprocess.run(["pdftotext", file_path, output_filename], check=True)
                log_event(f"Conversão PDF2TXT: {file.filename} -> {output_filename}")
                return send_from_directory(app.config['UPLOAD_FOLDER'], output_filename,
as_attachment=True)
            except subprocess.CalledProcessError as e:
                log_event(f"Erro na conversão PDF2TXT: {file.filename} - {e}")
                return f"Erro na conversão: {e}", 500
    return '''
    <!doctype html>
    <title>Upload PDF</title>
    <h1>Upload PDF para Conversão</h1>
    <form method="POST" enctype="multipart/form-data">
      <input type="file" name="file">
      <input type="submit" value="Upload">
    </form>
    '''

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0', port=5001)
```

5. Dockerfiles (Exemplo: pdf2txt/Dockerfile): (exemplo)

```
FROM python:3.9-slim-buster
```

```
WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

EXPOSE 5001

CMD ["python", "app.py"]
```

6. requirements.txt: (Para todos os serviços Web)

```
Flask
subprocess
```

7. docker-compose.yml: (certifique-se de que os nomes e portas estejam consistentes)

```
version: "3.9"
services:
  central_log:
    build: ./central_log
    ports:
      - "6000:6000"
    networks:
      - my-network

  main_app:
    build: ./main_app
    ports:
      - "5005:5005"
    networks:
      - my-network
    depends_on:
      - central_log
      - pdf2txt
      - reduce_resolution
      - resize_images
      - convert_images

  pdf2txt:
    build: ./pdf2txt
    ports:
      - "5001:5001"
    networks:
      - my-network
    depends_on:
      - central_log
```

```
  reduce_resolution:
    build: ./reduce_resolution
    ports:
      - "5002:5002"
    networks:
      - my-network
    depends_on:
      - central_log

  resize_images:
    build: ./resize_images
    ports:
      - "5003:5003"
    networks:
      - my-network
    depends_on:
      - central_log

  convert_images:
    build: ./convert_images
    ports:
      - "5004:5004"
    networks:
      - my-network
    depends_on:
      - central_log

networks:
  my-network:
```

Lembre-se de criar os arquivos para os outros serviços (`reduce_resolution`, `resize_images`, `convert_images`) de forma análoga ao exemplo `pdf2txt`. Adapte os códigos para as funcionalidades específicas de cada serviço, mantendo a inclusão do *timestamp* e o envio do relógio vetorial. Certifique-se de que o `num_processes` e `process_id` estejam configurados corretamente em cada serviço e na aplicação principal.

Recursos

Exemplos de Flask (upload e download):

- como fazer um upload de arquivo com Flask:
  - https://roytuts.com/python-flask-file-upload-example/
  - https://pythonbasics.org/flask-upload-file/
- upload e download:
  - https://docs.faculty.ai/user-guide/apis/flask_apis/flask_file_upload_download.html
  - https://www.geeksforgeeks.org/uploading-and-downloading-files-in-flask/

Tudo deve ser feito em Linux.

Entregáveis

Para esta versão, não será obrigatória a entrega do **relógio vetorial**. Cada aplicação poderá ter o seu próprio log local, registrando as informações locais apenas.
As demais funcionalidades deverão ser entregues.

- Relatório detalhado do projeto, incluindo todas as decisões de implementação bem como as evidências de execução (prints de tela).
- Todo o código, e instruções detalhadas de como colocar para rodar.
- Detalhamento da participação de cada componente do grupo no desenvolvimento do projeto.

Este projeto poderá ser feito em grupos de no máximo 3 (três) componentes.
Todo o código deverá ser entregue via atividade do **Classroom** do **GitHub**.