Distributed Music Editor

Relatório

Rafael Sousa Vilaça – 107476

Vitalie Bologa - 107854

universidade de aveiro

2022/2023

Introdução

O presente relatório descreve o trabalho desenvolvido no âmbito da disciplina de Computação Distribuída, um projeto relacionado com a *Advanced Sound Systems* (ASS).

O objetivo deste projeto foi criar um sistema inovador de karaoke, na qual essa aplicação permite remover não apenas a voz das músicas, mas também instrumentos individuais, permitindo que os músicos substituam as partes instrumentais da música. Disponibilizando através de um portal web, os músicos podem fazer upload das suas músicas, selecionar os instrumentos desejados e receber uma nova versão com apenas esses instrumentos.

O trabalho envolveu a análise dos requisitos da ASS, a definição de uma arquitetura distribuída, o desenvolvimento de um servidor e workers para processamento paralelo, bem como a implementação de um protocolo de comunicação eficiente.

Através deste projeto, buscamos fornecer uma solução técnica robusta e de alta qualidade para atender as demandas da ASS e proporcionar uma experiência única aos que utilizam aplicação.

Arquitetura

- RabbitMQ – a arquitetura é baseada em mensagens. O server é responsável pelo envia de mensagens para o RabbitMQ, que são guardadas temporariamente numa fila até que sejam consumidas pelas pelos workers. As mensagens são roteadas para as filas corretas por meio de exchanges e bindings, que definem as regras de roteamento. Um virtual host cria um ambiente isolado, garantindo a separação entre filas, exchanges e permissões. O broker é o servidor RabbitMQ que coordena a comunicação entre produtores e consumidores, assegurando a entrega confiável de mensagens. Essa arquitetura permite a comunicação assíncrona e escalável em sistemas distribuídos, possibilitando a troca eficiente de mensagens entre os componentes.
- FrontEnd – é baseado em templates em HTML, CSS e JavaScript. Os templates representam diferentes páginas e componentes da interface do utilizador. Eles são responsáveis por exibir informações relevantes, como formulários e resultados, e oferecer recursos interativos visando criar uma interface amigável e interativa para os utilizadores, melhorando a experiência de uso da aplicação.
- Client/Server – é responsável por receber dados do utilizador sobre o áudio desejado, como o ID da música e instrumentos selecionados. Em seguida, ele envia essas informações em uma mensagem para o *RabbitMQ* através da fila “task_queue”. O cliente aguarda a resposta do sistema, recebendo partes separadas do áudio processadas pelos *workers* e o áudio final composto, pelos instrumentos selecionados. Caso a música já tenha sido processada previamente, se houver alteração nos instrumentos selecionados o ficheiro final é construído no server com as respetivas *tracks*. A sua arquitetura é simples, focada em enviar solicitações e receber resultados processados, permitindo a interação do usuário com o sistema.
- Worker – atua como um consumidaor de mensagens e recebe tarefas de processamento de áudio enviadas pelo servidor. O worker utiliza o ThereadPoolExecutor para executar tarefas de forma assíncrona, sendo os resultados guardados em arquivos separados. Essa arquitetura permite uma distribuição eficiente

das tarefas de processamento de áudio, melhorando a escalabilidade e o desempenho do sistema como um todo.

- ThreadPoolExecutor – é utilizada para coordenar um conjunto de threads que executam tarefas em paralelo simultaneamente, aumentando a eficiência e o desempenho do sistema. O ThreadPoolExecutor recebe as tarefas do cliente e as distruibui entre as threads disponíveis na pool. Cada theread executa uma tarefa de forma independente, e quando concluída fica disponível para executar uma nova tarefa.
- Demucs – é um modelo de separação de arquivos de áudio baseado em redes neurais. A sua arquitetura consiste em um encoder, módulos de separação e decoder. O encoder processa o áudio de entrada. Os módulos de separação são responsáveis por separar cada fonte específica do áudio. Por fim o decoder reconstrói as fontes separadas em formate de áudio.

Protocolo

O protocolo utilizado neste projeto é o AMQP (Advanced Message Queuing Protocol). O AMQP é o protocolo base no qual foi construído o RabbitMQ. Este protocolo trabalha na camada de aplicação, determinando como devem ser transmitidos os dados ao longo do

```
message_header = {'music_id': music_id,
                  'piece_id': piece['id'],
                  'selected_instruments': music.instrument_tracks,
                  'job_id': job_id_counter}
```

sistema.

No contexto deste projeto o server e os worker utilizam o protocolo para trocar mensagens através da “task_queue” e “return_queue” do RabbitMQ. O server atua como um produtor/consumidor, onde ele publica as tarefas na fila. Os workers, por sua vez, são consumidores/produtores da fila, conectando-se ao RabbitMQ(broker) e aguardando por mensagens/tarefas. Quando uma mensagem é recebida, o worker processa a tarefa associada e envia a resposta de volta para o servidor através da uma fila de retorno.

O uso do AMQP traz benefícios importantes para o projeto, como a garantia de entrega confiável das mensagens, a capacidade de enfileirar as tarefas e distribuí-las aos workers disponíveis, a capacidade de lidar com alta carga de mensagens de forma escalável e lida com casos de falhas, como a desconexão do server ou worker.

Os Headers são utilizados para transmitir informações relevantes sobre as tarefas e mensagens trocadas entre o server e workers. Desta forma, é garantida a concordância entre o servidor e os workers (IMG LEG – Mensagem do servidor para começar o processamento de uma secção do áudio)

```
def return_message(ch, method, properties, body):
    global music_data
    music_id = properties.headers['music_id']
    piece_id = properties.headers['piece_id']
    track = properties.headers['track']
    print("music_id: ", music_id)
    print("piece_id: ", piece_id)
    print("track: ", track)

    job_id = properties.headers['job_id']
    print("job_id: ", job_id)
    size = properties.headers['size_file']
    print("size: ", size)
    time = properties.headers['processed_time']
    print("time: ", time)
```

O protocolo consiste em utilizar as propriedades da mensagem recebida para extrair informações específicas. Essas informações podem ser utilizadas para tomar decisões ou executar ações com base nas mesmas.

Fluxo do Sistema

- Music upload

```
@app.route('/music', methods=['POST','GET'])
def submit_music():
```

Esta route é acionada ao clicar no botão submit no frontend na mesma estão os métodos post e get no post adiciona-se a música com as respetivas informações ao server (nestas estão inclusos os dados e o ficheiro da música em si) de seguida tem o método get que apresenta as mesmas em conjunto com o html (allmusic.html)

- Music edit

```
@app.route('/music/edit/<int:music_id>', methods=['POST'])
def edit_music(music_id):
```

Esta route é utilizada ao clicar no botão editar com as novas tracks escolhidas, a mesma simplesmente modifica as tracks q já se encontravam selecionadas

- Music sended to process/music status

```
@app.route('/music/<int:music_id>', methods=['POST','GET'])
def process_music(music_id):
```

Esta route é utilizada ao escolher uma musica pelo seu id e ao clicar no botão submit ,se a musica desejada já tiver sido processada alguma vez apenas é refeito o file merged com as tracks desejadas no método Post e apresentadas as tracks desejadas (desta forma existe uma diminuição substancial em termos de tempo de execução) no método Get caso contrario é enviado ao rabbitmq (através da função send_message) a musica partida em 4 partes (esta divisão é efetuada na função split_music) com os respetivos headers para o /os workers processarem, sendo apresentado o site com o html (music.html) com o progresso atual , dado o processamento como concluído são apresentados os files para fazer download

```
def return_message(ch, method, properties, body):
```

A informação retornada pelo worker é processada nesta função ou seja, são guardadas as tracks nas suas respetivas localizações e os dados do job

```
def combine_music_pieces(track, music_id, num_pieces):
```

E quando a música se der como processada é efetuada a função combine_music_pieces para as partes serem unidas

- Music Download

```
@app.route('/audio/<path:filename>')
def serve_audio(filename):
    audio_path = 'CompleteAudio/' + filename
    return send_file(audio_path, mimetype='audio/mp3', as_attachment=True)
```

Esta route é a responsável pelo download dos files ao clicar no link dos mesmos no html (music.html)

- Reset System

```
@app.route('/reset', methods=['POST'])
def reset_system():
```

Esta route é executada no html (start.html) a mesma apaga todos os files de músicas e toda a informação nos queues do rabbitmq

- Job Status

```
@app.route('/job', methods=['GET'])
def list_jobs():
    return render_template('jobs.html', jobs=jobs)

@app.route('/job/<int:job_id>', methods=['GET'])
def get_job(job_id):
    #sem uso ja se encontra implementado em cima
    """ for job in jobs:
        if job.job_id == job_id:
            return jsonify(job)
    return jsonify({'message': 'Job não encontrado'}) """
```

Apesar de existir a route job<int:job_id> a mesma não é utilizada porque decidimos na route /job apresentar diretamente as informações sobre os jobs se o mesmo ainda não as tiver simplesmente aparecem vazias com o id correspondente, o ficheiro html é (jobs.html)

- Worker Message handling (Server é idêntico logo não será mencionado)

```
credentials = pika.PlainCredentials('guest','guest')
connection = pika.BlockingConnection(pika.ConnectionParameters(host='localhost',virtual_host='broker',credentials=credentials))
channel = connection.channel()

channel.queue_declare(queue='task_queue', durable=True)
print(' [*] Waiting for messages. To exit press CTRL+C')
```

```
channel.basic_qos(prefetch_count=1)
channel.basic_consume(queue='task_queue', on_message_callback=callback)
channel.start_consuming()
connection.close()
```

```
def callback(ch,method, properties, body):
    executor.submit(process_message(ch, method, properties, body))
```

O worker ao ser inicializado conecta-se ao rabbitmq no queue task_queue e fica à espera de mensagens para processar, esta conexão é feita numa thread separada para evitar o bloqueio da conexão para alem disso é feita a limitação de apenas uma mensagem ser recebida de cada vez através do prefetch_count

- Worker Processing

```
def process_message(ch, method, properties, body):
    print(" [x] Received ")
```

O worker começa o processamento da mensagem, o mesmo divide o áudio file nas tracks, envia as mesmas de volta e o merged_file desejado para o servidor através do rabbitmq.

A mensagem é enviada através desta função, a mesma cria uma nova thread para evitar o bloqueio da conexão e envia a mensagem para a queue return_message é importante mencionar q o ficheiro de áudio é enviado em binário , para tal é criado um objeto do tipo

```
send_message(merged_audio_data, message_header)
```

```
def send_message(data,header):
    executor.submit(send_msg(data, header))
```

```
def send_msg(data,header):
    stream = BytesIO(data)
    stream.seek(0)
    channel.queue_declare(queue='return_queue', durable=True)
    channel.basic_publish(
        exchange='',
        routing_key='return_queue',
        body=stream.read(),
        properties=pika.BasicProperties(
            delivery_mode=pika.spec.PERSISTENT_DELIVERY_MODE,
            headers=header
        )
    )
```

BytesIO (stream de bytes em memoria) e escolhida a posição da leitura da mesma como 0 (o inicio) sendo então enviada na mensagem essa stream com o respetivo header para identificação (para mais detalhe verificar secçao sobre o protocolo) por fim é escolhido o delivery_mode como persistent desta forma apenas quando o servidor mandar um acknowledgedé que a mensagem vai ser considerada como entregue desta forma previne-se perda de informação em caso de crash (etc..) do Servidor durante o processamento da mesma

```
ch.basic_ack(delivery_tag=method.delivery_tag)
```

O worker faz o mesmo sobre a mensagem enviada pelo server confirmando assim que conclui o processamento da mesma.

Conclusão

Em conclusão, este projeto de desenvolvimento de um sistema inovador de karaoke atingiu os seus objetivos com sucesso.

Melhoramos as nossas habilidades técnicas e de desenvolvimento ao lidar com a complexidade da arquitetura distribuída, implementando técnicas de processamento paralelo e integrar diferentes tecnologias.

No geral, este projeto nos proporcionou uma experiência de aprendizado, permitindo aplicar conceitos teóricos em um contexto prático.