AS 2021-2022

Projeto 2 - AS

Tema: Kafka

Autores:
João Ferreira nºmec 80041
João Magalhães nºmec 79923

Data: 14/05/2022

Conteúdo

1 Introdução

Este trabalho foi realizado no âmbito da Unidade Curricular Arquiteturas de Software do Mestrado em Engenharia Informática da Universidade de Aveiro e visa a implementação dos atributos de qualidade do *Apache Kafka*.

Para percebermos e implementarmos diferentes atributos de qualidade foram realizados diversos *use cases* e assim desenvolvidos diversos *producers* e diversos *consumers* para os diferentes *use cases*, de modo a cumprirmos os requisitos de cada um. Esses *use cases* foram definidos pelo docente e os requisitos de cada um estão presentes no enunciado deste trabalho.

A linguagem de programação utilizada neste projeto foi *JAVA* versão 11. É também necessária a instalação da biblioteca do Kafka que foi disponibilizada pelo docente na página da unidade curricular.

2 Principais Entidades

Para a implementação do projeto proposto, foi disponibilizados dados de um sensor, sendo que este documento permite simular o processamento de dados vindos de sensores com recurso a Kafka Clusters. Assim para cada *use case* foi necessário criar três principais entidades *PSource, PProducer, PConsumer*, as suas implementações dependem dos requisitos de cada *use case*.

- PSource lê os dados do ficheiro e envia-os via socket para o PProducer de acordo com os requisitos de cada UC.
- PProducer recebe os dados, e envia-os para o kafka cluster para que o PConsumer os possa ler.
- PConsumer consome os dados publicados.

As entidades referidas acima possuem interfaces própria. Existe ainda uma interface *main* para o utilizador escolher o *use case* que pretende executar.

3 *Use Cases*

3.1 UC1

Para o UC1, os *records* são lidos no PSOURCE e enviados para o PPRODUCER. No PPRODUCER, estes *records* são exibidos e enviados para o Kafka Cluster. No PCONSUMER, os records são lidos do Kafka cluster e exibidos. Este use case é limitado a um kafka producer e a um kafka consumer. O producer deve ter as propriedades com valor default.

Assim, para criarmos o producer definimos algumas propriedades. Após a consulta dos slides da UC, observamos que as propriedades ack, retries e max in flight requests per connection eram mencionadas. Optamos por definir estas propriedades com os valores default.

Para além disso, optamos também por consultar a documentação existente e escolhemos as propriedades que consideramos serem mais úteis para este use case. Estas propriedades também foram definidas com os valores default.

```
/* Producer properties mentioned in the course slides - values : default */
String ack = "all";
int retries = 2147483647;
int maxInFlightRequestsPerConnection = 5;

/* Some producer properties that we considered useful */
long bufferMemory = 33554432;
int batchSize = 16384;
int deliveryTimeoutMs = 120000;
```

Figura 1: UC1 - Propriedades do Kafka Producer

Para este use case, existem 3 processos: PSOURCE, PPRODUCER e PCONSUMER. No processo PSOURCE, é criada uma thread (a que chamamos sender) que irá enviar os dados via socket para outra thread (a que chamamos receiver), criada no PPRODUCER. A thread sender espera numa condição de um ReentrantLock e vai recebendo signal() para enviar a mensagem.

```
/* thread that will send data via socket initialization */
this.sender = new Thread(() -> {
    while(true){

        rl.lock();
        try {
            condition.await();
            threadSendToSocket( record );
        } catch (InterruptedException ex) {
            Logger.getLogger(PSource.class.getName()).log(Level.SEVERE, null, ex);
        } finally {
            rl.unlock();
        }
    }
});
```

Figura 2: UC1 - Thread que irá enviar as mensagem via socket

De seguida, é criada, no PPRODUCER, uma thread que irá receber estes dados e enviá-los para o Kafka cluster, recorrendo ao kafka producer que foi criado. O kafka producer é uma thread que fica bloqueada à espera de um signal() para que possa enviar cada record para o kafka cluster.

```
/* thread to receive data from psource */
Thread receiver;
receiver = new Thread(() -> {
    readDataAndUpdateGUI();
});

receiver.start();
```

Figura 3: UC1 - Thread que irá receber as mensagem via socket

```java
@Override
public void run(){
    while(true){
        this.rl.lock();
        try {
            this.condition.await();

            ProducerRecord<String, String> toSend = new ProducerRecord<>(this.topic, this.key, this.msg);

            this.producer.send(toSend);
        } catch (InterruptedException ex) {
            Logger.getLogger(kafkaProducer.class.getName()).log(Level.SEVERE, null, ex);
        } finally {
            this.rl.unlock();
        }
    }
}
```

Figura 4: UC1 - Ciclo de vida kafka producer

Por fim, no processo PCONSUMER é criada uma thread Kafka Consumer que irá ler os dados do cluster.

```java
this.consumer = new kafkaConsumer("localhost:9092, localhost:9093, "
        + "localhost:9094, localhost:9095, "
        + "localhost:9096, localhost:9097", "consumer",
        this);

this.consumer.subscribeTopic("Sensor");

this.consumer.start();
```

Figura 5: UC1 - Criação do kafka consumer

```java
@Override
public void run(){

    while(true){
        ConsumerRecords<String, String> records = consumer.poll( Duration.ofMillis(100) ); // read data from kafka

        for (ConsumerRecord<String, String> record : records) {
            this.pconsumer.updateSensor( record.value() ); // update gui from pconsumer

        }
    }
}
```

Figura 6: UC1 - Ciclo de vida kafka consumer

3.2 UC2

O use case 2 seguia a mesma lógica do use case 1 (assim como os restantes use cases). Para estes UC, o producer deveria minimizar a latência, podendo perder alguns dados. Deveriam ser criados 6 kafka producers.

O consumer poderia reprocessar os dados e deveria ter 6 kafka consumers.

Para o producer definimos as seguintes propriedades: acks, retries, max in flight requests per connection, linger ms e deliver timeout ms. Optamos por utilizar estas propriedades dado que foram as que resultaram da nossa pesquisa [1] [2].

```
/* Producer properties mentioned in the course slides - values : default */
int maxInFlightRequestsPerConnection = 5;

/**
 * Minimize latency
 * Source : https://docs.confluent.io/cloud/current/client-apps/optimizing/latency.html
*/
int lingerMs = 0;
String compressionType = "none";
String ack = "1";


 /**
  * Avoid data loss properties
  * Source: https://blog.softwaremill.com/help-kafka-ate-my-data-ae2e5d3e6576
*/

//String ack = "all";  // minimize latency > data loss
int retries = 2147483647;
int deliveryTimeoutMs = 2147483647;
```

Figura 7: UC2 - Propriedades dos kafka producers

Para o consumer definimos as seguintes propriedades enable auto commit, allow auto create topics, auto offset reset e fetch min bytes. Optamos por utilizar estas propriedades dado que foram as que resultaram da nossa pesquisa [4].

```
/**
 * Properties to avoid duplicates and data loss when commiting offsets
 * + consumer.commitAsync() -> low latency
 * Source: https://strimzi.io/blog/2021/01/07/consumer-tuning/
 */
boolean enableAutoCommit = false;
boolean allowAutoCreateTopics = false;
String autoOffsetReset = "earliest";
int fetchMinBytes = 10000;
```

Figura 8: UC2 - Propriedades dos kafka consumers

A lógica da troca de mensagens via socket era igual à do use case 1, com a diferença que agora tínhamos 6 threads a enviar (PSOURCE) e 6 threads a receber (PPRODUCER). As threads no PSOURCE seguem a mesma lógica do use case 1 (bloquear, desbloquear, enviar a mensagem, bloquear).

No PPRODUCER são criadas 6 threads que irão enviar as mensagens para 6 partições diferentes (no mesmo tópico - Sensor). A thread 1 enviará mensagens do sensor 1 para a partição 0 (o número das partições começa em 0) e assim sucessivamente.

```
@Override
public void run(){
    while(true){
        this.rl.lock();
        try {
            this.condition.await();

            ProducerRecord<String, String> toSend =
                    new ProducerRecord<>(this.topic, this.partition,
                            this.key, this.msg);

            this.producer.send(toSend);
        } catch (InterruptedException ex) {
        } finally {
            this.rl.unlock();
        }
    }
}
```

Figura 9: UC2 - Ciclo de vida das threads dos kafka producers

```
/**
 * Start Kafka producers
 */
this.producer1 = new kafkaProducer( servers, ack, retries,
        maxInFlightRequestsPerConnection, lingerMs, compressionType,
        deliveryTimeoutMs ) ;
this.producer1.start();

this.producer2 = new kafkaProducer( servers, ack, retries,
        maxInFlightRequestsPerConnection, lingerMs, compressionType,
        deliveryTimeoutMs) ;
this.producer2.start();

this.producer3 = new kafkaProducer( servers, ack, retries,
        maxInFlightRequestsPerConnection, lingerMs, compressionType,
        deliveryTimeoutMs) ;
this.producer3.start();

this.producer4 = new kafkaProducer( servers, ack, retries,
        maxInFlightRequestsPerConnection, lingerMs, compressionType,
        deliveryTimeoutMs) ;
this.producer4.start();

this.producer5 = new kafkaProducer( servers, ack, retries,
        maxInFlightRequestsPerConnection, lingerMs, compressionType,
        deliveryTimeoutMs) ;
this.producer5.start();

this.producer6 = new kafkaProducer( servers, ack, retries,
        maxInFlightRequestsPerConnection, lingerMs, compressionType,
        deliveryTimeoutMs) ;
this.producer6.start();
```

Figura 10: UC2 - Criação das threads dos kafka producers

No PCONSUMER são criadas 6 threads kafka consumer que irão ler as mensagens de 6 partições diferentes. A thread 1 irá ler mensagens da partição 0 (o número das partições começa em 0) e assim sucessivamente.

De modo a evitar dados repetidos, no momento da criação do PProducer e do PCONSUMER é-lhes passado um timestamp como argumento. Este timestamp irá ser utilizado para verificar se as mensagens pertencem à execução atual. Caso as mensagens pertençam a execuções anteriores são descartadas.

```
@Override
public void run(){

    while(true){
        ConsumerRecords<String, String> records = consumer.poll(
                Duration.ofMillis(30) );

        for (ConsumerRecord<String, String> record : records) {
            if ( record.value().split(":")[1].equals( Long.toString(this.time) ) ){
                this.pconsumer.updateSensor( record.value(), numSensor);
            }
        }

        /*
        Does not wait for the broker to respond to a commit request
        -> low latency
        source : https://strimzi.io/blog/2021/01/07/consumer-tuning/
        */
        consumer.commitAsync();

        /*
        commits the offsets of all messages returned from polling.
        -> high latency
        -> low throughput
        */
        //consumer.commitSync();
    }
}
```

Figura 11: UC2 - Ciclo de vida das threads dos kafka consumers

```
 * One per each Sensor
 */
this.consumer1 = new kafkaConsumer("localhost:9092, localhost:9093, "
        + "localhost:9094, localhost:9095, localhost:9096, "
        + "localhost:9097", "consumer", this, fetchMinBytes,
        enableAutoCommit, allowAutoCreateTopics,
        autoOffsetReset, 1, this.time); // number of sensor, time of mes
this.consumer1.subscribeTopic("Sensor", 0); // topic, partition
this.consumer1.start();

this.consumer2 = new kafkaConsumer("localhost:9092, localhost:9093, "
        + "localhost:9094, localhost:9095, localhost:9096, "
        + "localhost:9097", "consumer", this, fetchMinBytes,
        enableAutoCommit, allowAutoCreateTopics,
        autoOffsetReset, 2, this.time); // number of sensor, time of mes
this.consumer2.subscribeTopic("Sensor", 1); // topic, partition
this.consumer2.start();

this.consumer3 = new kafkaConsumer("localhost:9092, localhost:9093, "
        + "localhost:9094, localhost:9095, localhost:9096, "
        + "localhost:9097", "consumer", this, fetchMinBytes,
        enableAutoCommit, allowAutoCreateTopics,
        autoOffsetReset, 3, this.time); // number of sensor, time of mes
this.consumer3.subscribeTopic("Sensor", 2); // topic, partition
this.consumer3.start();

this.consumer4 = new kafkaConsumer("localhost:9092, localhost:9093, "
        + "localhost:9094, localhost:9095, localhost:9096, "
        + "localhost:9097", "consumer", this, fetchMinBytes,
        enableAutoCommit, allowAutoCreateTopics,
        autoOffsetReset, 4, this.time); // number of sensor, time of mes
this.consumer4.subscribeTopic("Sensor", 3); // topic, partition
this.consumer4.start();

this.consumer5 = new kafkaConsumer("localhost:9092, localhost:9093, "
        + "localhost:9094, localhost:9095, localhost:9096, "
        + "localhost:9097", "consumer", this, fetchMinBytes,
        enableAutoCommit, allowAutoCreateTopics,
        autoOffsetReset, 5, this.time); // number of sensor, time of mes
this.consumer5.subscribeTopic("Sensor", 4); // topic, partition
this.consumer5.start();

this.consumer6 = new kafkaConsumer("localhost:9092, localhost:9093, "
        + "localhost:9094, localhost:9095, localhost:9096, "
        + "localhost:9097",
        "consumer", this, fetchMinBytes,
        enableAutoCommit, allowAutoCreateTopics,
        autoOffsetReset, 6, this.time); // number of sensor, time of mes
this.consumer6.subscribeTopic("Sensor", 5); // topic, partition
this.consumer6.start();
```

Figura 12: UC2 - Criação das threads dos kafka consumers

3.3 UC3

Para o use case 3 era necessário criar apenas 3 kafka producers, maximizando o throughput e minimizando a possibilidade de perder dados. Em relação ao PCONSUMER, este deve criar um consumer group com 3 kafka consumers.

Para maximizar o throughput utilizamos as seguintes propriedades [3] [4] [5]:

- batch size - producer;
- linger ms - producer;

- compression type - producer;
- ack - producer;
- buffer memory - producer;
- fetch min bytes - consumer;
- fetch max wait ms- consumer;

Para minimizar a possibilidade de perder dados [1] [4]:

- retries - producer;
- delivery timeout ms - producer;
- enable auto commit - ambos;
- allow auto create topics - producer;
- auto offset reset - ambos;
- fetch min bytes - ambos;

```
/* Producer properties mentioned in the course slides - values : default */
int maxInFlightRequestsPerConnection = 5;

/**
 * Maximum throughput
 * Source: https://docs.confluent.io/cloud/current/client-apps/optimizing/throughput.html
 * https://granulate.io/optimizing-kafka-performance/
 * https://strimzi.io/blog/2021/01/07/consumer-tuning/
 */
int batchSize = 200000;
int lingerMs = 100;
String compressionType = "lz4";
String ack = "1";
int bufferMemory = 50000000;

this.gui = new PProducerGUI("", 0, 250);
this.gui.setVisible(true);


/**
 * Properties to avoid data loss when commiting offsets
 * Source: https://strimzi.io/blog/2021/01/07/consumer-tuning/
 * https://blog.softwaremill.com/help-kafka-ate-my-data-ae2e5d3e6576
 */
//String ack = "all"; // maximum throughput > data loss
int retries = 2147483647;
int deliveryTimeoutMs = 2147483647;

boolean enableAutoCommit = false;
boolean allowAutoCreateTopics = false;
String autoOffsetReset = "earliest";
int fetchMinBytes = 10000;
```

Figura 13: UC3 - Propriedades dos kafka producers

```java
/**
 * Properties to avoid duplicates and data loss when commiting offsets
 * + consumer.commitAsync() -> low latency
 * Source: https://strimzi.io/blog/2021/01/07/consumer-tuning/
 */
boolean enableAutoCommit = false;
boolean allowAutoCreateTopics = false;
String autoOffsetReset = "earliest";

/**
 * Maximum throughput
 * Source : https://docs.confluent.io/cloud/current/client-apps/optimizing/throughput.html
 * https://granulate.io/optimizing-kafka-performance/
 * https://strimzi.io/blog/2021/01/07/consumer-tuning/
 */
int fetchMinBytes = 100000;
int fetchMaxWaitMs = 500;
```

Figura 14: UC3 - Propriedades dos kafka consumers

Para este use case definimos apenas 3 partições. Cada um dos 3 kafka producers irá enviar dados para a partição correspondente. O primeiro producer ficará responsável por enviar os dados do sensor 1 e do sensor 4 para a partição 0, o segundo producer ficará responsável por enviar os dados do sensor 2 e sensor 5 para a partição 1 e o terceiro producer ficará responsável por enviar os dados do sensor 3 e sensor 6 para a partição 2. Por fim, os kafka consumers irão ler dados de cada uma destas partições. Sendo que também são apenas 3, o primeiro consumer irá ler os dados da primeira partição (sensores 1 e 4), o segundo consumer irá ler os dados da segunda partição (sensores 2 e 5) e o terceiro consumer irá ler os dados da terceira partição (sensores 3 e 6). Os ciclos de vida dos kafka producers e consumers são semelhantes aos anteriores. De modo a tentarmos minimizar a possibilidade de perder records definimos propriedades como enable auto commit (false), allow auto create topics (false) e auto offset reset (”earliest”).

3.4 UC4

Para a implementação do use case 4, reutilizamos o código desenvolvido no use case 2. Para além dos 6 kafka producers, optamos também por criar 6 kafka consumers. De modo a tentarmos minimizar a possibilidade de perder records definimos propriedades como ack (all), retries (2147483647), delivery timeout ms (2147483647), enable auto commit (false) e auto offset reset (”earliest”).

3.5 UC5

Para a implementação do use case 5 era necessário criar 6 partições e 3 consumer groups com 3 consumers cada.Era necessário também disponibilizar informação sobre a temperatura mínima e máxima, seguindo a tática Voting Replication.

Dado que iríamos criar 3 consumers por cada grupo, optamos por criar também 3 producers. Cada um dos producers iria escrever para uma das partições de acordo com o tipo de record. No caso do record ser do sensor 1 ou 4, este seria enviado pelo producer 1, sendo que records do sensor 1 iriam ser enviados para a partição 0 e do sensor 4 para a partição 3. No caso do record ser do sensor 2 ou 5, este seria enviado pelo

producer 2, sendo que records do sensor 2 iriam ser enviados para a partição 1 e do sensor 5 para a partição 4. No caso do record ser do sensor 3 ou 6, este seria enviado pelo producer 3, sendo que records do sensor 3 iriam ser enviados para a partição 2 e do sensor 6 para a partição 5.

Foram ainda definidos 3 consumer groups (group1, group2, group3), cada um com 3 kafka consumers. Cada um destes consumers iria ler os dados de duas partições diferentes (consumer 1 - partições 0 e 3; consumer 2- partições 1 e 4; consumer 3 - partições 2 e 5). Cada consumer group possui uma GUI independente dos outros 2 grupos. Cada grupo de consumers irá calcular a temperatura mínima e máxima individualmente e, posteriormente, propagar o valor de cada record para o PCONSUMER. Caso o valor deste record for menor que o mínimo, então o mínimo passa a ser igual a este valor. Caso o valor deste record for maior que o máximo, então o máximo passa a ser igual a este valor. Sempre que os valores do mínimo ou do máximo são alterados, a interface é atualizada. Foi criada uma gui auxiliar que permite visualizar o valor mínimo e máximo da temperatura (o menor e maior valor de acordo com todos os grupos).

3.6 UC6

Para a implementação do use case 6 era necessário criar 6 partições e 1 consumer group com 3 consumers. Era necessário também disponibilizar informação sobre a temperatura média. Dado que iríamos criar 3 consumers por cada grupo, optamos por criar também 3 producers. Cada um dos producers iria escrever para uma das partições de acordo com o tipo de record. No caso do record ser do sensor 1 ou 4, este seria enviado pelo producer 1, sendo que records do sensor 1 iriam ser enviados para a partição 0 e do sensor 4 para a partição 3. No caso do record ser do sensor 2 ou 5, este seria enviado pelo producer 2, sendo que records do sensor 2 iriam ser enviados para a partição 1 e do sensor 5 para a partição 4. No caso do record ser do sensor 3 ou 6, este seria enviado pelo producer 3, sendo que records do sensor 3 iriam ser enviados para a partição 2 e do sensor 6 para a partição 5.

Os 3 consumers iriam contar o número de records lidos e incrementar o valor dos records lidos, de modo a tornar possível o cálculo da média. A média é disponibilizada na interface dos consumers.

4 Como Executar o projeto

Para executar este projeto é necessário executar o script para o *use case* que se pretende. Os scripts encontram-se no directório *scripts* e se pretender executar o *use case 1* é necessário executar o ficheiro *./uc1.sh*, por exemplo. Posteriormente, é necessário executar a classe *Main.java* que se encontra na pasta Main, na raiz do projeto. A execução desta classe irá disponibilizar a interface necessária para executar cada *use case* e posteriormente é necessário que o utilizador selecione o *use case* que pretende nessa interface. Finalmente, para terminar a utlização do kafka do *use case* utilizado é necessário executar o script *./stop.sh* que se encontra também no directório *scripts*.

Exemplo de execução:

- Executar no terminal ./uc1.sh (directório *scripts*);
- Exexcutar a classe Main.java (directório *Main*);
- Na interface, selecionar o *UC1*;
- No final, executar no terminal ./stop.sh (directório *scripts*);

5 Limitações

Apesar de termos desenvolvido todos os use cases, alguns deles têm algumas limitações:

- por vezes, o consumer do use case 1 demora alguns segundos a iniciar a leitura dos dados, provocando perda de dados.
- (UC1) por vezes, ocorrem erros na construção das interfaces gráficas.
- não conseguimos minimizar a possibilidade de um kafka consumer falhar (use case 6);

Excluindo estes três pontos, tudo o resto funciona.

6 Contribuições

Consideramos que ambos os elementos do grupo contribuíram de forma equivalente para o desenvolvimento deste projeto. Assim, a contribuição de cada elemento foi igual a 50% do trabalho desenvolvido. As tarefas realizadas por cada elemento estão evidenciadas na tabela abaixo.

Contribuição	Autor
UC1	João Ferreira
UC2	João Magalhães
UC3	João Ferreira
UC4	João Magalhães
UC5	João Ferreira
UC6	João Magalhães
Relatório	Ambos

7 Resultados

Dentro do directório do projeto, existe um directório com screenshots de execuções para um ficheiro txt com cerca de 7500 linhas. Todos os use cases lêem o ficheiro sensor.txt por default (PSOURCE).

8 Conclusão

A realização deste trabalho permitiu-nos adquirir conhecimentos de Kafka, bem como perceber alguns dos seus atributos. Consideramos ter alcançado todos os objetivos propostos com sucesso, nomeadamente de cada use case.

Referências

[1] How to prevent data loss in kafka. https://blog.softwaremill.com/help-kafka-ate-my-data-ae2e5d3e6576. Accessed: 2022-05-15.

[2] Optimizing for latency. https://docs.confluent.io/cloud/current/client-apps/optimizing/latency.html. Accessed: 2022-05-15.

[3] Optimizing for throughput. https://docs.confluent.io/cloud/current/client-apps/optimizing/throughput.html. Accessed: 2022-05-15.

[4] Optimizing kafka consumers. https://strimzi.io/blog/2021/01/07/consumer-tuning/. Accessed: 2022-05-15.

[5] Optimizing kafka performance. https://granulate.io/optimizing-kafka-performance/. Accessed: 2022-05-15.