Sisteme Distribuite - Laborator 10

Aplicaţii flux şi monitorizare utilizând Apache Kafka

Apache Kafka - descriere generală

Kafka este un sistem de mesagerie de tip *publish / subscribe*, descris ca un „*commit log* distribuit”, sau o „platformă de *streaming* distribuită”. Faţă de alte sisteme de acest tip, Kafka a fost proiectat ca datele să fie stocate pe termen lung, iar acestea pot fi citite în mod determinist şi în ordinea în care au fost scrise.

Unitatea de bază cu care funcţionează sistemul Kafka este **mesajul** (eng. ***message***). Ca şi concept, este similar cu o înregistrare într-un tabel dintr-o bază de date. Mesajele pot avea metadate folosite pentru identificare, numite **chei** (eng. ***keys***). Cheile sunt utilizate pentru scrierea mesajelor în **partiţii**, fapt care asigură un control mai granular.

Mesajele în Kafka sunt organizate pe **subiecte** (eng. ***topics***). Acestea pot fi asemănate cu un folder în sistemul de fişiere sau un tabel într-o bază de date. La rândul lor, subiectele sunt împărţite în **partiţii** (eng. ***partitions***). O partiţie este echivalentă cu un *commit log*. Partiţiile reprezintă o modalitate prin care Kafka oferă redundanţă şi scalabilitate.

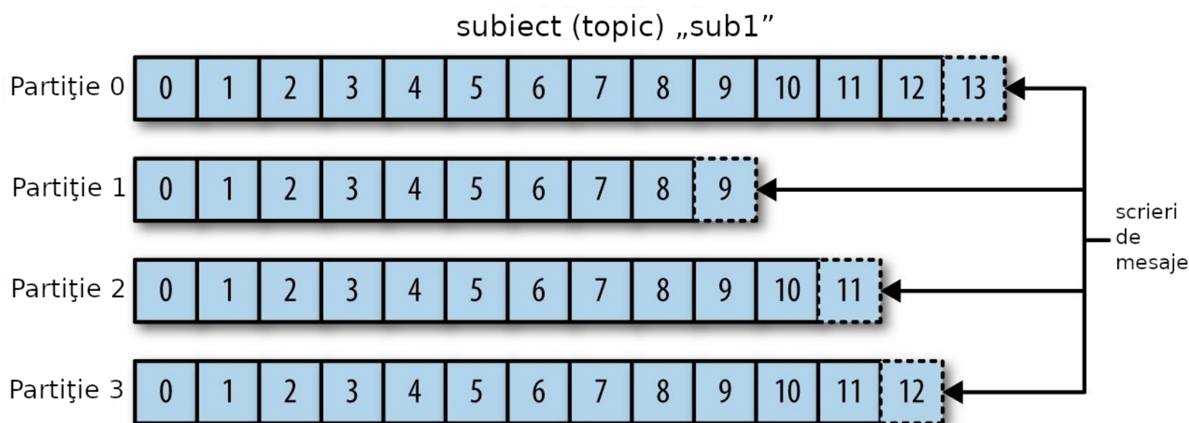


Figura 1 - Subiect Kafka cu 4 partiţii

Entităţile componente utilizate în laborator, precum şi relaţiile dintre acestea sunt scoase în evidenţă în următoarea diagramă:

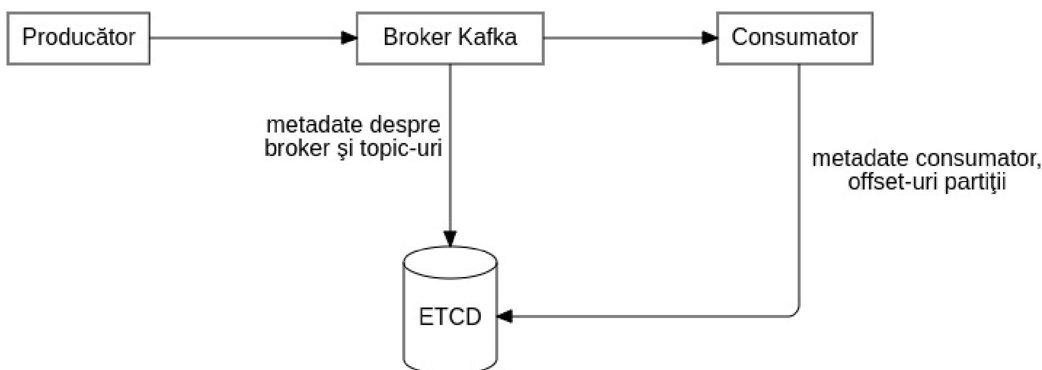


Figura 2 - Funcţionarea Kafka şi ETCD

ETCD

ETCD este un depozit distribuit de tip cheie-valoare, care asigură o consistenţă sporită şi furnizează o modalitate sigură de a stoca date ce trebuie accesate de un sistem distribuit sau de un cluster de maşini de calcul. De asemenea, gestionează automat alegerile de lider din timpul partiţionărilor reţelei şi poate tolera defecte ale maşinilor de calcul, chiar şi la nivelul nodului lider.

Datele sunt stocate în directoare organizate ierarhic, asemenea unui sistem de fişiere clasic. Sistemul de stocare persistă mai multe versiuni ale datelor simultan: atunci când o valoare este înlocuită de una nouă, se păstrează versiunea anterioară a acesteia. Depozitul cheie-valoare este imutabil, operaţiile nu actualizează structura de date, ci mereu generează o altă structură actualizată, având posibilitatea de a accesa şi versiunile anterioare ale valorilor corespunzătoare cheilor.

Valorile scrise într-un depozit ETCD pot fi urmărite pentru modificări, astfel încât aplicaţiile care le folosesc să fie notificate pentru eventuale reconfigurări.

Atenţie! Cei care lucrează de pe stațiile din laborator, săriți direct la Secțiunea “Pornire server Kafka”.

Instalare şi configurare Apache Kafka

Pe lângă server-ul Kafka propriu-zis, trebuie instalat şi un serviciu de configurare distribuit, pe care Kafka îl utilizează pentru a stoca metadatele *broker*-ilor: nodurile componente din cluster, *topic*-urile, partiţiile etc.

Aveţi nevoie de **JRE versiunea 8** şi de **Docker Engine** instalate pe sistemul dvs. pentru ca paşii următori să funcţioneze corect.

Pornire serviciu de configurare ETCD

Pentru laborator, s-a ales sistemul de stocare distribuit de tip cheie-valoare **ETCD** (https://github.com/etcd-io/etcd).

Mai întâi, creaţi folder-ele în care ETCD îşi va stoca fişierele temporare, respectiv datele şi asignaţi-i permisiunile corecte:

```
sudo mkdir /var/etcd-data /var/etcd-data.tmp
sudo chgrp docker /var/etcd-data /var/etcd-data.tmp
sudo chmod g=rwx /var/etcd-data /var/etcd-data.tmp
```

Apoi, utilizând Docker, porniţi o instanţă local de ETCD, cu următoarea comandă (**dacă nu doriţi repornirea automată a containerului, eliminaţi parametrul marcat cu roşu**):

```
docker run -d --restart=always -p 2379:2379 -p 2380:2380 --mount
type=bind,source=/var/etcd-data.tmp,destination=/var/etcd-data --name
etcd-gcr-v3.4.7 gcr.io/etcd-development/etcd:v3.4.7
/usr/local/bin/etcd --name s1 --data-dir /var/etcd-data --listen-
client-urls http://0.0.0.0:2379 --advertise-client-urls
http://0.0.0.0:2379 --listen-peer-urls http://0.0.0.0:2380 --initial-
advertise-peer-urls http://0.0.0.0:2380 --initial-cluster
s1=http://0.0.0.0:2380 --initial-cluster-token tkn --initial-cluster-
state new --log-level info --logger zap --log-outputs stderr
```

Folosiţi comanda **`docker logs`** ca să verificaţi funcţionarea serviciului ETCD. Preluaţi ID-ul container-ului Docker returnat de comanda anterioară (sau folosiţi comanda **`docker ps`**) şi executaţi:

```
docker logs <ID_CONTAINER_ETCD>
```

Un exemplu de output este cel din figura următoare:

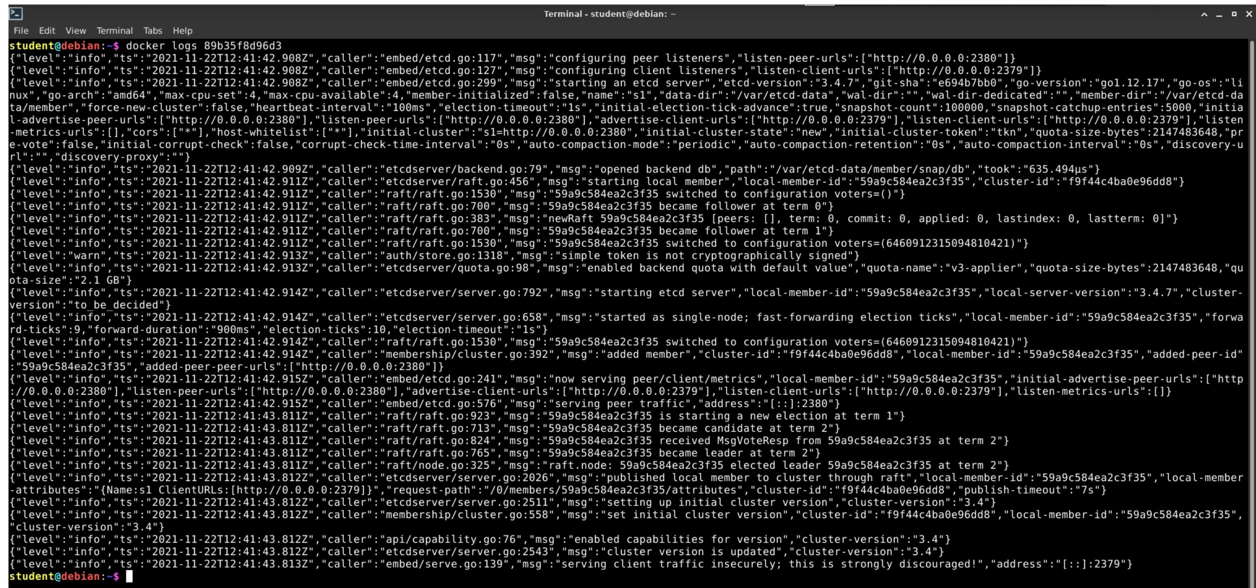


Figura 3 - Verificarea funcţionării serviciului ETCD containerizat

Compilare Kafka cu suport ETCD

Aveţi nevoie de utilitarul `gradle` instalat pentru a putea compila Kafka.
(se poate instala pe Debian 10 folosind comanda: **`sudo apt install gradle`**)
Se clonează *repository*-ul GitHub:

```
git clone https://github.com/banzaicloud/apache-kafka-on-k8s.git
```

Dacă *repository*-ul nu mai este valabil, puteți găși o copie a acestuia în folderul *Resources* din codul exemplu ataşat.

Se modifică fişierul **`build.gradle`**, adăugând un parametru nou în vectorul **`scalaCompileOptions.additionalParameters`** de pe linia **325**:

```
"-target:jvm-1.8"
```

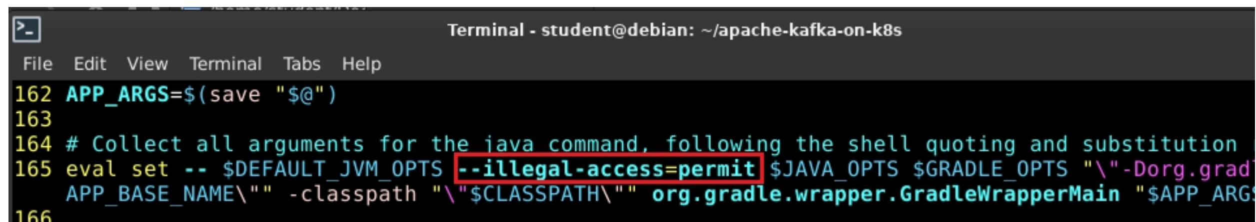
În folderul **`apache-kafka-on-k8s`** se execută comanda:

```
gradle
```

Apoi se compilează proiectul cu comanda:

```
./gradlew releaseTarGz -x signArchives
```

În cazul în care obţineţi eroarea: “Unrecognized option: --illegal-access=permit”, deschideţi fişierul gradlew şi ştergeţi “--illegal-access=permit” de pe linia 165:



Rezultatul compilării este o arhivă numită **`kafka_2.11-2.0.0-SNAPSHOT.tgz`**, care se regăseşte în folder-ul **`core/build/distributions`**, relativ la folder-ul proiectului.

Instalare Apache Kafka

Folosiţi versiunea modificată de Apache Kafka, compilată cu paşii de mai sus, care poate utiliza ETCD cu rol de serviciu de configurare. Se consideră că aveţi arhiva **`kafka_2.11-2.0.0-SNAPSHOT.tgz`** obţinută în urma compilării Kafka.

Dezarhivaţi arhiva:

```
tar -xzvf kafka_2.11-2.0.0-SNAPSHOT.tgz
```

Copiaţi conţinutul în folder-ul **`/opt`**:

```
sudo mv kafka_2.11-2.0.0-SNAPSHOT /opt/kafka
```

Asignaţi-vă utilizatorul şi grupa ca şi proprietari ai folder-ului **`kafka`**:

```
sudo chown -R $USER:$(groups | awk '{ print $1 }') /opt/kafka
```

Configurare Kafka pentru utilizarea ETCD ca serviciu de configurare

Deschideţi fişierul **`/opt/kafka/config/server.properties`** cu un editor de text şi modificaţi proprietatea **`metastore.connect`** astfel încât să aibă valoarea **`etcd://localhost:2379`**.

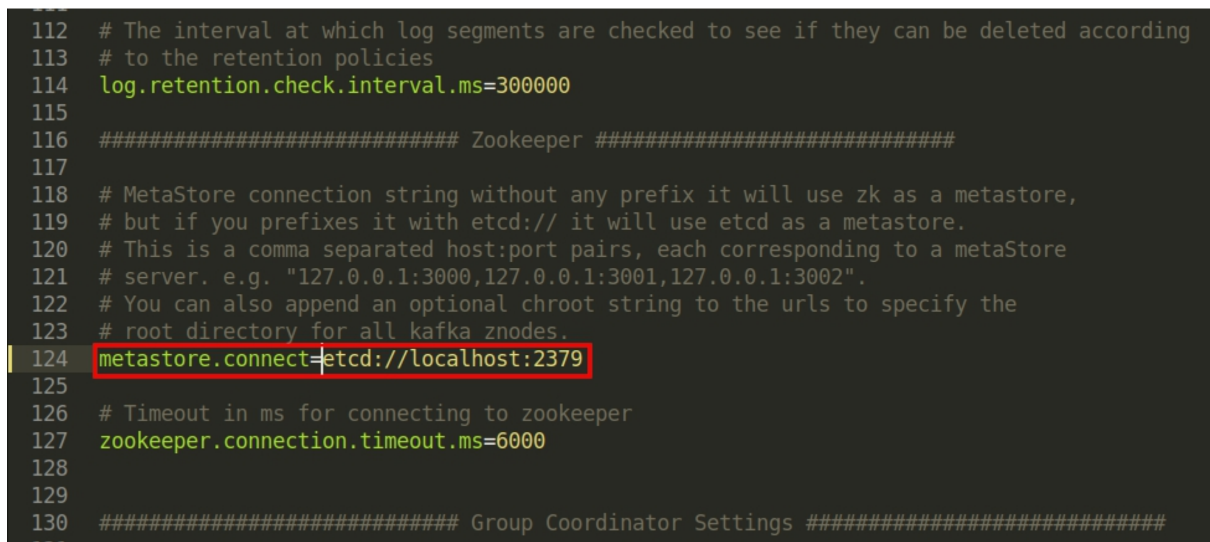


Figura 4 - Setarea ETCD ca serviciu de configurare Kafka

Pornire server Kafka

Nu uitaţi să aveţi pornit containerul Docker cu ETCD înainte de a porni Kafka!

Server-ul Apache Kafka poate fi pornit astfel:

```
chmod +x /opt/kafka/bin/*.sh
sudo -E /opt/kafka/bin/kafka-server-start.sh -daemon
/opt/kafka/config/server.properties
```

(comanda nu afişează nimic după execuţie)

Implicit, la rularea comenzii sudo, sunt resetate toate variabilele de environment (ex.

JAVA_HOME). Pentru prezervarea acestora, la comanda sudo se folosește argumentul -E. La unele din comenzile următoare, dacă nu se specifică argumentul respectiv, veți avea erori de tipul "java not found" sau "JAVA_HOME not set".

Se verifică apoi pornirea corectă a server-ului:

```
ps -ef | grep kafka
```

Dacă server-ul a pornit fără erori, comanda anterioară va returna un proces Java cu mulţi parametri în linie de comandă:

```
Terminal - student@debian: /opt/kafka/bin
File Edit View Terminal Tabs Help
student@debian:/opt/kafka/bin$ ps -ef | grep kafka
root       88641       1  3 10:20 pts/4    00:00:04 /usr/lib/jvm/jdk1.8.0_301/bin/java -Xmx1G -Xms1G -server -XX:+UseG1GC -XX:MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=35 -
XX:+ExplicitGCInvokesConcurrent -Djava.awt.headless=true -Xloggc:/opt/kafka/bin/../logs/kafkaServer-gc.log -verbose:gc -XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintGCTimeStamps
-XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=10 -XX:GCLogFileSize=100M -Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremot
e.ssl=false -Dkafka.logs.dir=/opt/kafka/bin/../logs -Dlog4j.configuration=file:/opt/kafka/bin/../config/log4j.properties -cp /opt/kafka/bin/../libs/activation-1.1.1.jar:/opt/kafka/bin/.
./libs/aopalliance-repackaged-2.5.0-b32.jar:/opt/kafka/bin/../libs/argparse4j-0.7.0.jar:/opt/kafka/bin/../libs/audience-annotations-0.5.0.jar:/opt/kafka/bin/../libs/commons-lang3-3.5.ja
r:/opt/kafka/bin/../libs/connect-api-2.0.0-SNAPSHOT.jar:/opt/kafka/bin/../libs/connect-file-2.0.0-SNAPSHOT.jar:/opt/kafka/bin/../libs/connect-json-2.0.0-SNAPSHOT.jar:/opt/kafka/bin/../l
ibs/connect-runtime-2.0.0-SNAPSHOT.jar:/opt/kafka/bin/../libs/connect-transforms-2.0.0-SNAPSHOT.jar:/opt/kafka/bin/../libs/error_prone_annotations-2.1.2.jar:/opt/kafka/bin/../libs/grpc-
context-1.9.1.jar:/opt/kafka/bin/../libs/grpc-core-1.9.1.jar:/opt/kafka/bin/../libs/grpc-netty-1.9.1.jar:/opt/kafka/bin/../libs/grpc-protobuf-1.9.1.jar:/opt/kafka/bin/../libs/grpc-proto
buf-lite-1.9.1.jar:/opt/kafka/bin/../libs/grpc-stub-1.9.1.jar:/opt/kafka/bin/../libs/gson-2.7.jar:/opt/kafka/bin/../libs/guava-19.0.jar:/opt/kafka/bin/../libs/guava-20.0.jar:/opt/kafka/
bin/../libs/hk2-api-2.5.0-b32.jar:/opt/kafka/bin/../libs/hk2-locator-2.5.0-b32.jar:/opt/kafka/bin/../libs/hk2-utils-2.5.0-b32.jar:/opt/kafka/bin/../libs/instrumentation-api-0.4.3.jar:/o
pt/kafka/bin/../libs/jackson-annotations-2.9.5.jar:/opt/kafka/bin/../libs/jackson-core-2.9.5.jar:/opt/kafka/bin/../libs/jackson-databind-2.9.5.jar:/opt/kafka/bin/../libs/jackson-jaxrs-b
ase-2.9.5.jar:/opt/kafka/bin/../libs/jackson-jaxrs-json-provider-2.9.5.jar:/opt/kafka/bin/../libs/jackson-module-jaxb-annotations-2.9.5.jar:/opt/kafka/bin/../libs/javassist-3.20.0-GA.ja
r:/opt/kafka/bin/../libs/javassist-3.21.0-GA.jar:/opt/kafka/bin/../libs/javax.annotation-api-1.2.jar:/opt/kafka/bin/../libs/javax.inject-1.jar:/opt/kafka/bin/../libs/javax.inject-2.5.0-
b32.jar:/opt/kafka/bin/../libs/javax.servlet-api-3.1.0.jar:/opt/kafka/bin/../libs/javax.ws.rs-api-2.0.1.jar:/opt/kafka/bin/../libs/jaxb-api-2.3.0.jar:/opt/kafka/bin/../libs/jersey-clien
t-2.25.1.jar:/opt/kafka/bin/../libs/jersey-common-2.25.1.jar:/opt/kafka/bin/../libs/jersey-container-servlet-2.25.1.jar:/opt/kafka/bin/../libs/jersey-container-servlet-core-2.25.1.jar:/
opt/kafka/bin/../libs/jersey-guava-2.25.1.jar:/opt/kafka/bin/../libs/jersey-media-jaxb-2.25.1.jar:/opt/kafka/bin/../libs/jersey-server-2.25.1.jar:/opt/kafka/bin/../libs/jetcd-common-0.0
.2.jar:/opt/kafka/bin/../libs/jetcd-core-0.0.2.jar:/opt/kafka/bin/../libs/jetcd-resolver-0.0.2.jar:/opt/kafka/bin/../libs/jetty-client-9.2.24.v20180105.jar:/opt/kafka/bin/../libs/jetty-
continuation-9.2.24.v20180105.jar:/opt/kafka/bin/../libs/jetty-http-9.2.24.v20180105.jar:/opt/kafka/bin/../libs/jetty-io-9.2.24.v20180105.jar:/opt/kafka/bin/../libs/jetty-security-9.2.2
4.v20180105.jar:/opt/kafka/bin/../libs/jetty-server-9.2.24.v20180105.jar:/opt/kafka/bin/../libs/jetty-servlet-9.2.24.v20180105.jar:/opt/kafka/bin/../libs/jetty-servlets-9.2.24.v20180105
.jar:/opt/kafka/bin/../libs/jetty-util-9.2.24.v20180105.jar:/opt/kafka/bin/../libs/jopt-simple-5.0.4.jar:/opt/kafka/bin/../libs/jsr305-3.0.0.jar:/opt/kafka/bin/../libs/kafka_2.11-2.0.0-
SNAPSHOT.jar:/opt/kafka/bin/../libs/kafka_2.11-2.0.0-SNAPSHOT-sources.jar:/opt/kafka/bin/../libs/kafka_2.11-2.0.0-SNAPSHOT-test-sources.jar:/opt/kafka/bin/../libs/kafka-clients-2.0.0-SN
APSHOT.jar:/opt/kafka/bin/../libs/kafka-log4j-appender-2.0.0-SNAPSHOT.jar:/opt/kafka/bin/../libs/kafka-streams-2.0.0-SNAPSHOT.jar:/opt/kafka/bin/../libs/kafka-streams-examples-2.0.0-SNA
PSHOT.jar:/opt/kafka/bin/../libs/kafka-streams-scala_2.11-2.0.0-SNAPSHOT.jar:/opt/kafka/bin/../libs/kafka-streams-test-utils-2.0.0-SNAPSHOT.jar:/opt/kafka/bin/../libs/kafka-tools-2.0.0-
SNAPSHOT.jar:/opt/kafka/bin/../libs/log4j-1.2.17.jar:/opt/kafka/bin/../libs/lz4-java-1.4.1.jar:/opt/kafka/bin/../libs/maven-artifact-3.5.3.jar:/opt/kafka/bin/../libs/metrics-core-2.2.0.
jar:/opt/kafka/bin/../libs/netty-buffer-4.1.17.Final.jar:/opt/kafka/bin/../libs/netty-codec-4.1.17.Final.jar:/opt/kafka/bin/../libs/netty-codec-http2-4.1.17.Final.jar:/opt/kafka/bin/../
libs/netty-codec-http-4.1.17.Final.jar:/opt/kafka/bin/../libs/netty-codec-socks-4.1.17.Final.jar:/opt/kafka/bin/../libs/netty-common-4.1.17.Final.jar:/opt/kafka/bin/../libs/netty-handle
r-4.1.17.Final.jar:/opt/kafka/bin/../libs/netty-handler-proxy-4.1.17.Final.jar:/opt/kafka/bin/../libs/netty-resolver-4.1.17.Final.jar:/opt/kafka/bin/../libs/netty-transport-4.1.17.Final
.jar:/opt/kafka/bin/../libs/opencensus-api-0.10.0.jar:/opt/kafka/bin/../libs/opencensus-contrib-grpc-metrics-0.10.0.jar:/opt/kafka/bin/../libs/osgi-resource-locator-1.0.1.jar:/opt/kafka
/bin/../libs/plexus-utils-3.1.0.jar:/opt/kafka/bin/../libs/protobuf-java-3.5.1.jar:/opt/kafka/bin/../libs/protobuf-java-util-3.5.1.jar:/opt/kafka/bin/../libs/proto-google-common-protos-
1.0.0.jar:/opt/kafka/bin/../libs/reflections-0.9.11.jar:/opt/kafka/bin/../libs/rocksdbjni-5.7.3.jar:/opt/kafka/bin/../libs/scala-library-2.11.12.jar:/opt/kafka/bin/../libs/scala-logging
_2.11-3.8.0.jar:/opt/kafka/bin/../libs/scala-reflect-2.11.12.jar:/opt/kafka/bin/../libs/slf4j-api-1.7.25.jar:/opt/kafka/bin/../libs/slf4j-log4j12-1.7.25.jar:/opt/kafka/bin/../libs/snapp
y-java-1.1.7.1.jar:/opt/kafka/bin/../libs/validation-api-1.1.0.Final.jar:/opt/kafka/bin/../libs/zkclient-0.10.jar:/opt/kafka/bin/../libs/zookeeper-3.4.12.jar kafka.Kafka /opt/kafka/conf
ig/server.properties
student    88822   85385  0 10:22 pts/4    00:00:00 grep kafka
student@debian:/opt/kafka/bin$
```

Figura 5 - Verificarea funcţionării server-ului Kafka

Testare broker Kafka

În continuare, testaţi **crearea unui subiect (*topic*)**:

```
sudo -E /opt/kafka/bin/kafka-topics.sh --create --zookeeper
etcd://localhost:2379 --replication-factor 1 --partitions 1 --topic
test
```

Verificaţi *topic*-ul creat astfel:

```
sudo -E /opt/kafka/bin/kafka-topics.sh --zookeeper
etcd://localhost:2379 --describe --topic test
```

Testaţi producerea unor mesaje *dummy* pentru *topic*-ul creat mai sus:

```
sudo -E /opt/kafka/bin/kafka-console-producer.sh --broker-list
localhost:9092 --topic test
```

Comanda de mai sus va aştepta să introduceţi, linie cu linie, câte un mesaj. Introduceţi câte unul, după care apăsaţi ENTER. Când aţi terminat, apăsaţi CTRL+D.

Testaţi acum consumarea mesajelor de test produse mai sus:

```
sudo -E /opt/kafka/bin/kafka-console-consumer.sh --bootstrap-server
localhost:9092 --topic test --from-beginning
```

Comanda va prelua mesajele de test anterioare şi le va afişa pe rând la consolă. Pentru a elibera consola şi a ieşi din procesul consumator de mesaje, apăsaţi CTRL+C.

Pentru a şterge un subiect (*topic*), puteţi folosi comanda:

```
sudo -E /opt/kafka/bin/kafka-topics.sh --delete --zookeeper
etcd://localhost:2379 --topic NUME_TOPIC
```

Atenţie: ştergerea unui *topic* este de tip *lazy*. Nu are efect imediat!

Nu uitaţi că, atât pe parcursul laboratorului, cât şi atunci când se lucrează cu aplicaţiile din laborator, aveţi nevoie de instanţa containerizată de ETCD şi de server-ul Apache Kafka pornite!

Exemplu de aplicaţie Kafka utilizând Python

Creaţi un proiect Python, iar în mediul virtual instalaţi modulul **`kafka-python`**, deoarece interfaţarea cu server-ul Kafka se face utilizând API-ul expus de acest modul (https://github.com/dpkp/kafka-python).

```
python3 -m venv exemplu_kafka_env
source ./exemplu_kafka_env/bin/activate
pip3 install kafka-python
```

Dacă doriţi să executaţi codul Python direct din PyCharm, se poate instala modulul **`kafka-python`** astfel: **File → Settings → Project: NUME_PROIECT → Python Interpreter** → apăsaţi pe semnul plus din dreapta (**Install**) şi căutaţi „**`kafka-python`**". Apăsaţi pe „**Install Package**" după ce aţi selectat modulul din lista de rezultate.

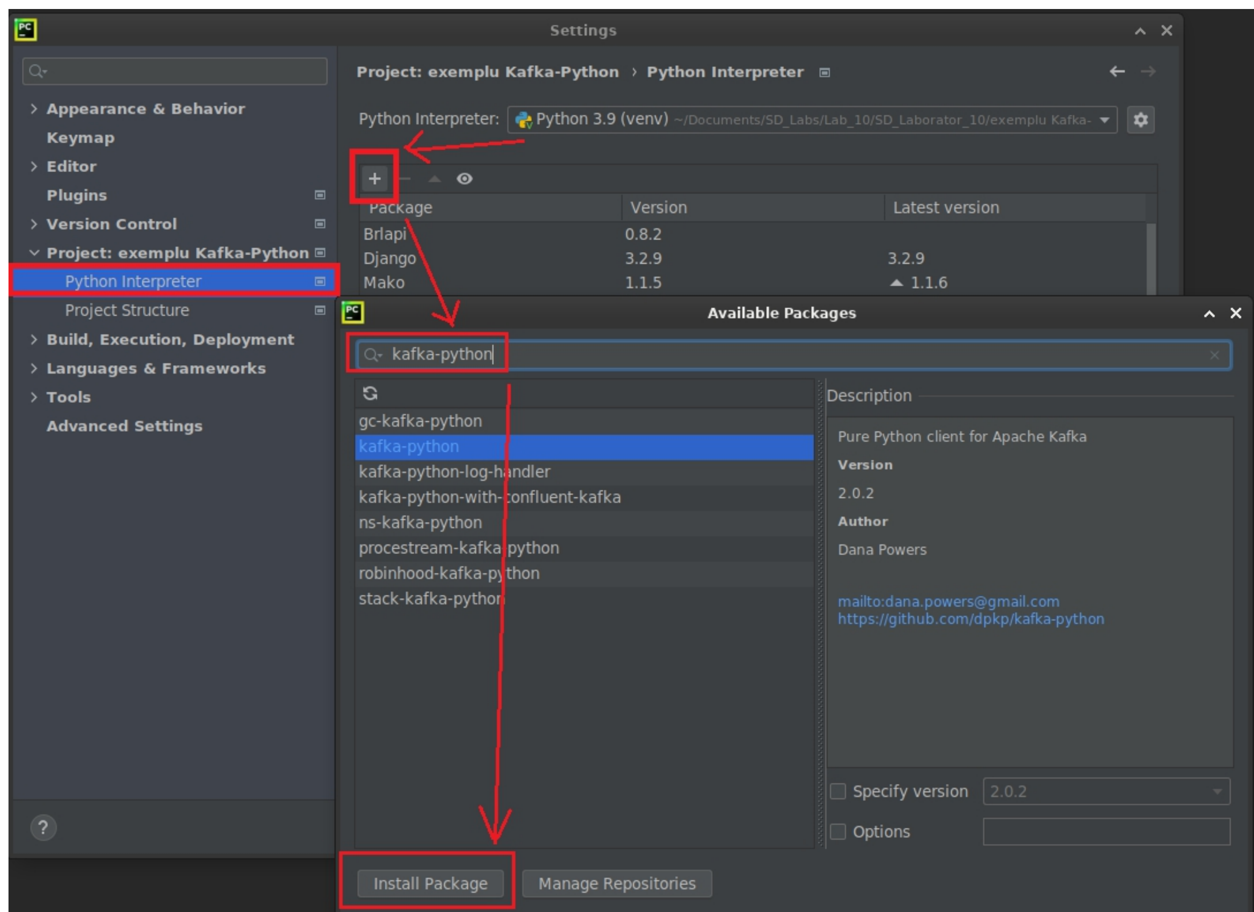


Figura 6 - Instalarea modulului **`kafka-python`** din PyCharm

Următorul cod exemplu foloseşte Kafka pentru a trimite mesaje între 2 entităţi: un *thread*

producător de mesaje şi un *thread* consumator de mesaje. Cele 2 *thread*-uri comunică prin *topic*-ul **`topic_exemplu_python`**. Funcţionarea aplicaţiei este redată în următoarea diagramă de activitate:

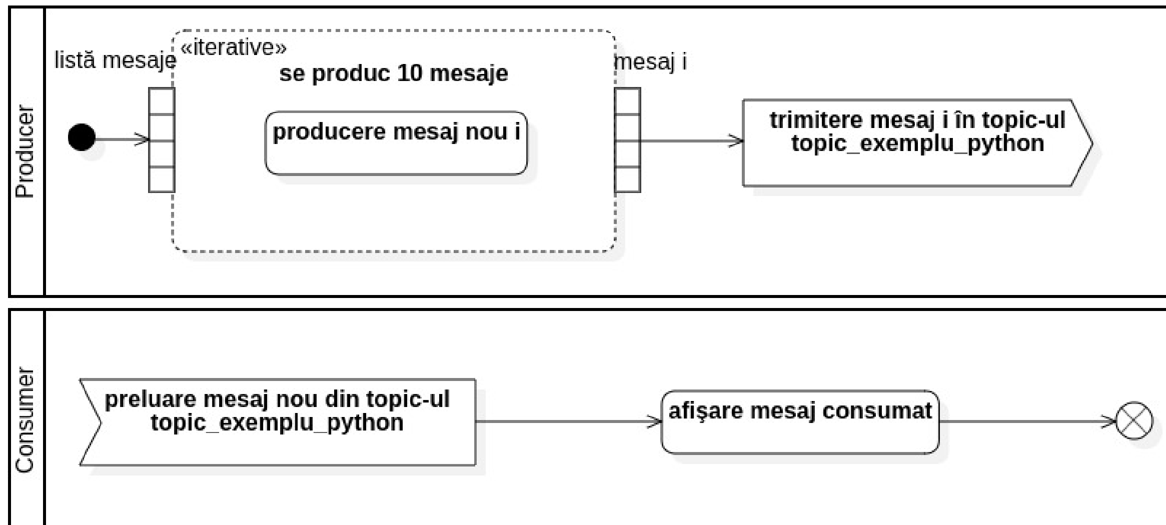


Figura 7 - Funcţionarea aplicaţiei exemplu de tip producător-consumator

- **exemplu.py**

```
from kafka import KafkaConsumer
from kafka import KafkaProducer
import threading


class Consumer(threading.Thread):
    def __init__(self, topic):
        super().__init__()
        self.topic = topic

    def run(self) -> None:
        consumer = KafkaConsumer(self.topic,
auto_offset_reset='earliest')
        # topicul va fi creat automat, daca nu exista deja

        # thread-ul consumator primeste mesajele din topic
        for msg in consumer:
            print("Am consumat mesajul: " + str(msg.value,
encoding="utf-8"))


class Producer(threading.Thread):
    def __init__(self, topic):
        super().__init__()
        self.topic = topic

    def run(self) -> None:
        producer = KafkaProducer()
        for i in range(10):
            message = 'mesaj {}'.format(i)

            # thread-ul producator trimite mesaje catre un topic
```

```
            producer.send(topic=self.topic, value=bytearray(message,
encoding="utf-8"))
            print("Am produs mesajul: {}".format(message))

        # metoda flush() asigura trimiterea batch-ului de mesaje
produse
        producer.flush()


if __name__ == '__main__':
    # se creeaza 2 thread-uri: unul producator de mesaje si celalalt
consumator
    producer_thread = Producer("topic_exemplu_python")
    consumer_thread = Consumer("topic_exemplu_python")

    producer_thread.start()
    consumer_thread.start()

    producer_thread.join()
    consumer_thread.join()
```

Încercaţi să executaţi aplicaţia de 2-3 ori. Ce observaţi că afişează *thread*-ul consumator de mesaje? De ce apar şi mesajele anterioare?

Exemplu de aplicaţie Kafka utilizând Kotlin

Aplicaţia Kotlin va expune un *endpoint* web care răspunde la o cerere HTTP de tip PUT pe calea **`/publish`**. Pentru fiecare cerere HTTP primită, se preia corpul cererii, iar acesta va fi produs sub formă de mesaj Kafka în *topic*-ul **`topic_exemplu_kotlin`**.

Consumatorul este o componentă Spring separată, sub formă de *listener* Kafka. Atunci când în *topic*-ul menţionat apare un mesaj nou, *listener*-ul consumă respectivul mesaj şi aşteaptă în continuare alte mesaje.

Creaţi un proiect **Spring Boot** (**ca în laboratorul 3**), **fără să adăugaţi dependenţa Spring Web**.

Adăugaţi dependenţa **Spring Kafka** astfel:

- **pentru Maven**:

```
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
    <version>2.5.16.RELEASE</version>
</dependency>
```

- **pentru Gradle**:

```
compile group: 'org.springframework.kafka', name: 'spring-kafka',
version: '2.5.16.RELEASE'
```

https://mvnrepository.com/artifact/org.springframework.kafka/spring-kafka

Atenţie la versiune: `2.5.16.RELEASE` funcţionează corect, având în vedere combinaţiile de versiuni Kafka şi Spring utilizate în laborator.

- **Producer.kt**

```
package com.sd.laborator

import org.springframework.kafka.core.KafkaTemplate
import org.springframework.stereotype.Component

@Component
class KotlinProducer(private val kafkaTemplate: KafkaTemplate<String,
String>) {
    fun send(message: String) {
        kafkaTemplate.send("topic_exemplu_kotlin", message)
    }
}
```

- **Consumer.kt**

```
package com.sd.laborator

import org.springframework.kafka.annotation.KafkaListener
import org.springframework.stereotype.Component

@Component
class KotlinConsumer {
    @KafkaListener(topics = ["topic_exemplu_kotlin"], groupId =
"exemplu-consumer-kotlin")
    fun processMessage(message: String) {
        println("Am consumat urmatorul mesaj: $message")
    }
}
```

- **KafkaController.kt**

```
package com.sd.laborator

import org.springframework.http.HttpStatus
import org.springframework.http.ResponseEntity
import org.springframework.stereotype.Controller
import org.springframework.web.bind.annotation.RequestBody
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RequestMethod
import org.springframework.web.bind.annotation.ResponseBody

@Controller
class KafkaController(private val kotlinProducer: KotlinProducer) {
    @RequestMapping(path = ["/publish"], method = [RequestMethod.PUT])
    @ResponseBody
    fun publishMessage(@RequestBody message: String):
ResponseEntity<HttpStatus> {
        kotlinProducer.send(message)
        return ResponseEntity(HttpStatus.CREATED);
    }
}
```

- **KafkaApp.kt**

```
package com.sd.laborator

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication

@SpringBootApplication
class SpringKafkaApplication

fun main(args: Array<String>) {
    runApplication<SpringKafkaApplication>(*args)
}
```

Compilaţi, împachetaţi şi executaţi aplicaţia Spring Boot.
Pentru testare, puteţi trimite cereri HTTP de tip PUT astfel:

```
curl -X PUT http://localhost:8080/publish --header "Content-Type: text/plain" --data-raw "mesaj test 1"
```

Alternativ, puteţi folosi aplicaţia **Postman**.
Ar trebui să găsiţi în log-ul aplicaţiei Spring Boot un mesaj de la componenta **`Consumer`**, de tipul celui din figură:

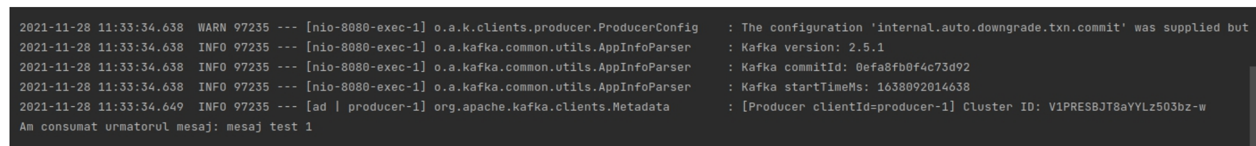


Figura 8 - Testarea aplicaţiei Kafka cu Kotlin

Kafka sub formă de broker de mesaje pentru arhitecturi bazate pe flux - procesarea fluxurilor

Se reia aplicaţia „Okazii” din **laboratorul 7**, care modelează funcţionarea unei sesiuni de licitaţie. De această dată, **comunicarea dintre microserviciile participante la licitaţie se face printr-un *broker* de mesaje Kafka**.

Proiectarea iniţială a aplicaţiei Okazii folosind Kafka pentru comunicare

Aşadar, diagrama de microservicii a aplicaţiei **Okazii** din laboratorul 7 se modifică astfel, la prima vedere:

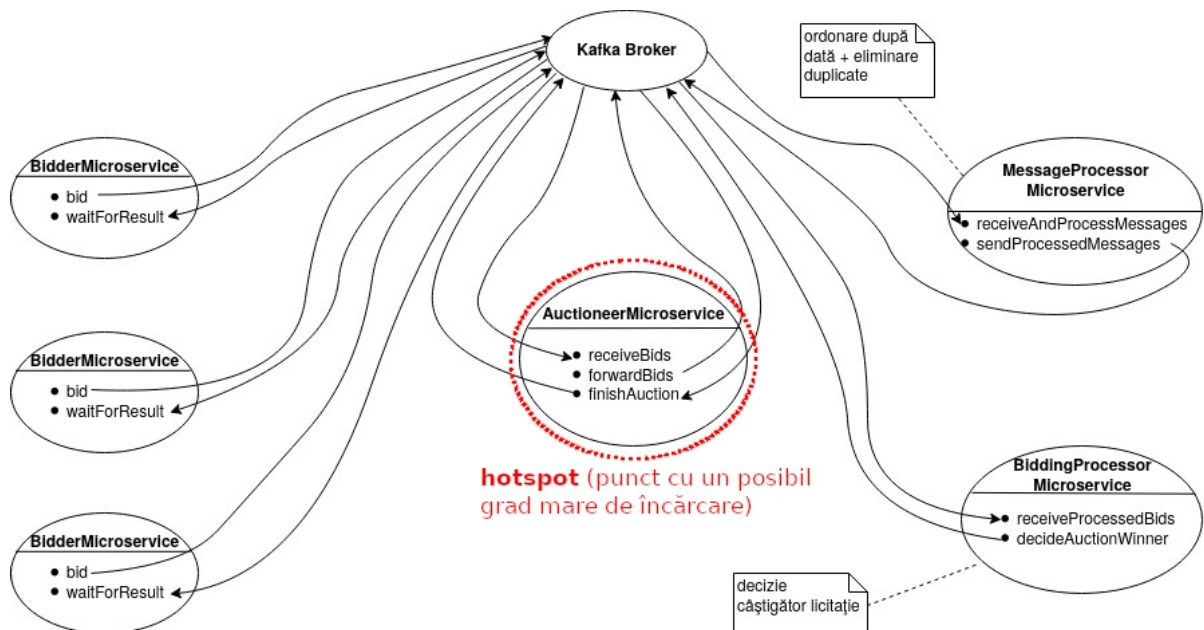


Figura 9 - Diagrama modificată de microservicii a aplicaţiei Okazii (cu *broker* Kafka)

Se observă că toată comunicaţia se face prin intermediul Kafka. Microserviciile se folosesc de *topic*-uri pentru a transmite mesaje în mod asincron şi decuplat de destinatarii acestora. S-a marcat pe diagramă **microserviciul care ar putea avea un nivel mare de încărcare**; atunci când numărul de entităţi **`Bidder`** creşte, numărul de mesaje primite de **`Auctioneer`** va fi prea mare pentru a putea fi procesat de o singură instanţă.

Se pot face însă câteva observaţii pentru îmbunătăţirea şi reproiectarea aplicaţiei:

- Odată ce un mesaj este consumat dintr-un *topic*, el rămâne stocat până în momentul în care expiră perioada definită de **politica de retenţie** (implicit, mesajele sunt şterse după 168 de ore de la publicare). Aşadar, faptul că **`AuctioneerMicroservice`** consumă un mesaj primit de la o entitate **`Bidder`** nu determină distrugerea imediată a mesajului din modelul de persistenţă Kafka.

```
95   ############################# Log Retention Policy #############################
96
97   # The following configurations control the disposal of log segments. The policy can
98   # be set to delete segments after a period of time, or after a given size has accumulated.
99   # A segment will be deleted whenever *either* of these criteria are met. Deletion always happens
100  # from the end of the log.
101
102  # The minimum age of a log file to be eligible for deletion due to age
103  log.retention.hours=168
```

Figura 10 - Politica de retenţie implicită a Kafka (**`/opt/kafka/config/server.properties`**)

- Fiecare consumator are câte un *offset* în cadrul unei partiţii dintr-un *topic*. Utilizând acel *offset*, consumatorul poate citi mesajele de câte ori doreşte, în ce direcţie doreşte. Aşadar, **microservicii diferite pot citi din acelaşi *topic* în mod independent, fiecare având propriul *offset***.

- Kafka permite ca mesajele să fie preluate simultan de mai mulţi consumatori. Se pot defini astfel aşa-numitele **grupuri de consumatori** (eng. *consumer groups*). Acest lucru ajută la eliminarea redundanţei, în sensul că: spre deosebire de modalitatea de procesare a mesajelor din laboratorul 7, microserviciul **`MessageProcessor`** îşi poate prelua singur setul de mesaje pe care trebuie să le proceseze **direct din acelaşi topic în care le primeşte**

microserviciul `Auctioneer` (consecinţă a observaţiei anterioare).

– Deci, nu mai este nevoie ca microserviciul **`Auctioneer`** să redirecţioneze mesajele primite către microserviciul **`MessageProcessor`**.

– Totodată, microserviciul **`BiddingProcessor`** nu trebuie să primească efectiv setul procesat de mesaje, ci trebuie doar să citească ce a publicat **`MessageProcessor`** într-un alt *topic* specific mesajelor „curate” (fără duplicate, ordonate după dată şi oră)

• Kafka asigură o **echilibrare minimală a încărcării** prin scalarea pe orizontală a consumatorilor. Asignând fiecărui consumator un identificator în cadrul grupei (*group ID*), Kafka foloseşte un algoritm de tip ***Round Robin*** pentru a citi mesajele în mod echilibrat, în funcţie de numărul de consumatori din grupă. Acest lucru asigură şi **toleranţa la defecte**, din punct de vedere al consumatorilor: dacă un microserviciu consumator se defectează, **mesajele vor fi redistribuite în funcţie de celelalte entităţi active din grupul de consumatori** şi de numărul de partiţii în care a fost împărţit *topic*-ul care este folosit. **Mesajele nu se pierd în caz de defecte! Se redistribuie.**

– de exemplu: fie un topic separat în 4 partiţii, la care se înscrie un grup de 2 consumatori. Primul şi al doilea mesaj vor fi primite de **consumatorul 1**, al treilea şi al patrulea vor fi primite de **consumatorul 2**, următoarele 2 mesaje vor fi primite din nou de **consumatorul 1** ş.a.m.d.

Reproiectarea aplicaţiei Okazii

În continuare, se reproiectează aplicaţia, aplicându-se proprietăţile şi avantajele oferite de Kafka, conform cu observaţiile anterioare.

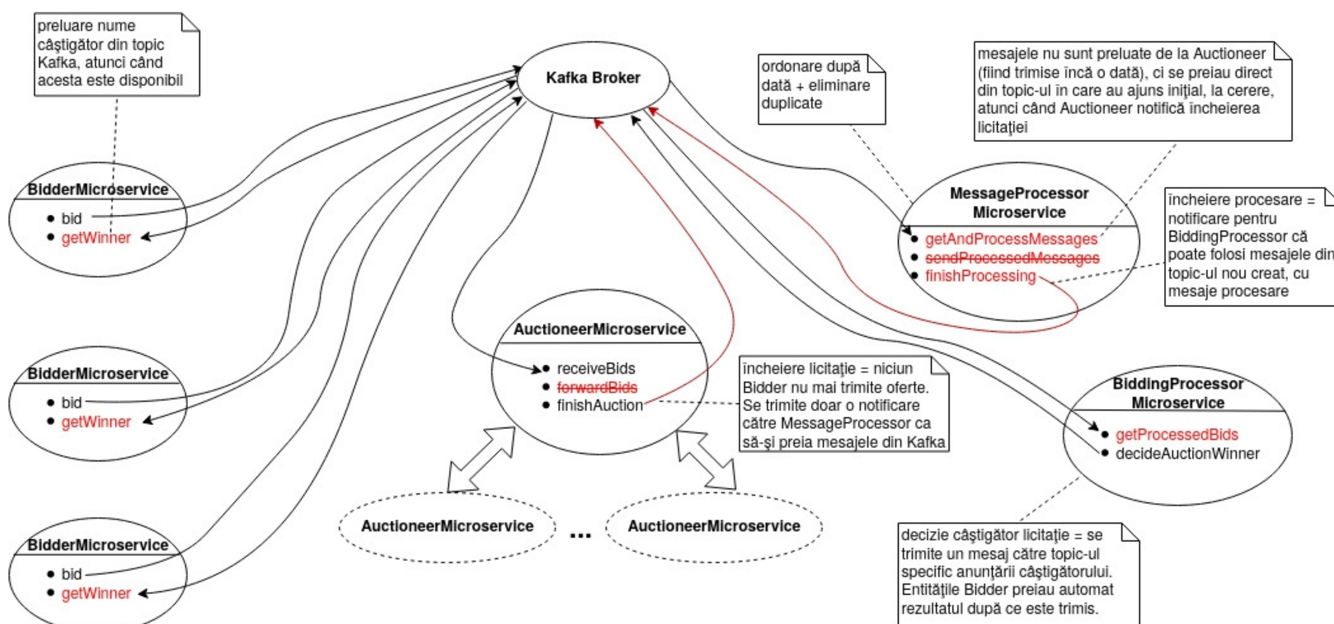


Figura 11 - Diagrama de microservicii reproiectată

S-au marcat pe diagramă modificările aduse microserviciilor. De asemenea, s-a scos în evidenţă posibilitatea de scalare a microserviciului **`Auctioneer`**.

Implementarea microserviciilor

Aplicaţia va fi implementată în Python, folosind modulul **`kafka-python`** pentru comunicarea cu *broker*-ul Kafka. Diagrama de clase este prezentată în figura următoare:

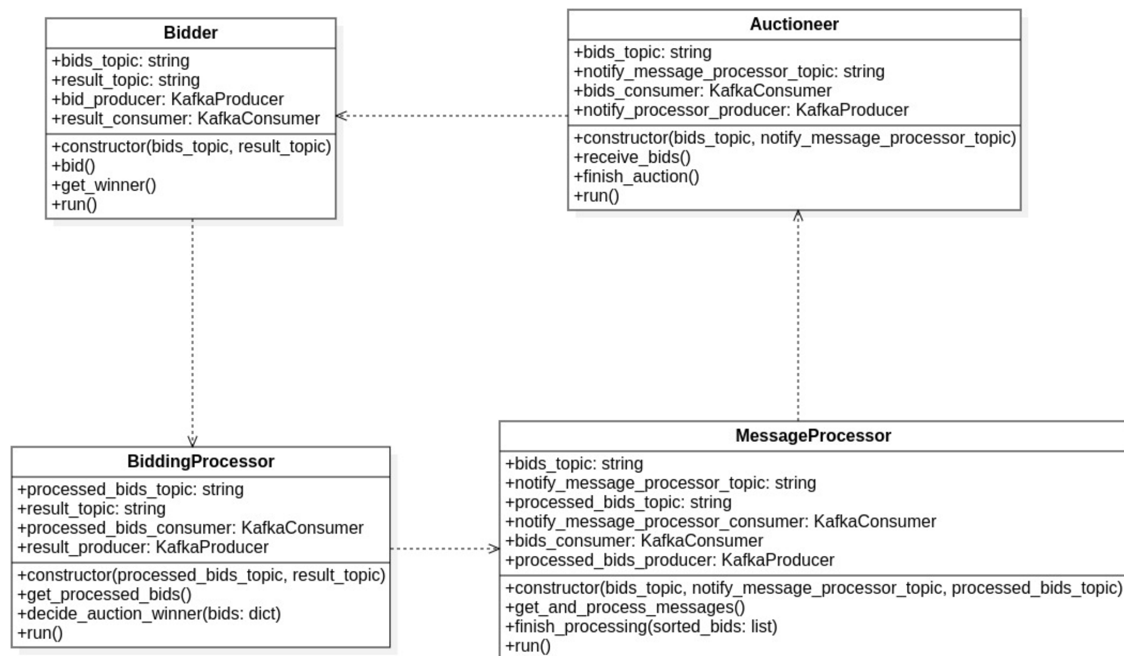


Figura 12 - Diagrama de clase aplicaţie Okazii

Relaţiile de dependenţă provin de la necesitatea de comunicare prin broker-ul de mesaje. Aşadar, aceste relaţii s-ar putea traduce prin „**are nevoie să comunice cu ...**”, sau „**aşteaptă un mesaj de la...**”.

Creaţi un proiect Python în PyCharm şi adăugaţi fişierele sursă menţionate în continuare.

- **Bidder.py**

```
from kafka import KafkaProducer, KafkaConsumer
from random import randint
from uuid import uuid4


class Bidder:
    def __init__(self, bids_topic, result_topic):
        super().__init__()
        self.bids_topic = bids_topic
        self.result_topic = result_topic

        # producatorul pentru oferte de licitatie
        self.bid_producer = KafkaProducer()

        # consumatorul pentru rezultatul licitatiei
        self.result_consumer = KafkaConsumer(
            self.result_topic,
            auto_offset_reset="earliest"  # mesajele se preiau de la
cel mai vechi la cel mai recent
        )

        self.my_bid = randint(1000, 10_000)  # se genereaza oferta ca
numar aleator intre 1000 si 10.000
```

```
        self.my_id = uuid4()  # se genereaza un identificator unic
pentru ofertant

    def bid(self):
        # se construieste mesajul pentru licitare
        print("Trimit licitatia mea: {}...".format(self.my_bid))
        bid_message = bytearray("licitez", encoding="utf-8")  # corpul
contine doar mesajul "licitez"
        bid_headers = [  # antetul mesajului contine identitatea
ofertantului si, respectiv, oferta sa
            ("amount", self.my_bid.to_bytes(2, byteorder='big')),
            ("identity", bytes("Bidder {}".format(self.my_id),
encoding="utf-8"))
        ]

        # se trimite licitatia sub forma de mesaj catre Kafka
        self.bid_producer.send(topic=self.bids_topic,
value=bid_message, headers=bid_headers)

        # exista o sansa din 2 ca oferta sa fie trimisa de 2 ori
pentru a simula duplicatele
        if randint(0, 1) == 1:
            self.bid_producer.send(topic=self.bids_topic,
value=bid_message, headers=bid_headers)

        self.bid_producer.flush()
        self.bid_producer.close()

    def get_winner(self):
        # se asteapta raspunsul licitatiei
        print("Astept rezultatul licitatiei...")
        result = next(self.result_consumer)

        # se verifica identitatea castigatorului
        for header in result.headers:
            if header[0] == "identity":
                identity = str(header[1], encoding="utf-8")

        if identity == self.my_id:
            print("Am castigat!!!")
        else:
            print("Am pierdut...")

        self.result_consumer.close()

    def run(self):
        self.bid()
        self.get_winner()


if __name__ == '__main__':
    bidder = Bidder(
        bids_topic="topic_oferte",
        result_topic="topic_rezultat"
    )
    bidder.run()
```

Microserviciul **`Bidder`** se foloseşte de 2 *topic*-uri Kafka:

– **`topic_oferte`** → aici îşi va publica oferta (generată aleator) sub formă de mesaj Kafka. Mesajul este de forma:

- corp: şirul de caractere „**`licitez`**”
- metadate:
  - **`amount`**: **`PRETUL_LICITAT`**
  - **`identity`**: **`IDENTIFICATOR_OFERTANT`**

– **`topic_rezultat`** → din acest *topic* va consuma un singur mesaj: rezultatul licitaţiei care va apărea în momentul în care **`BiddingProcessor`** va stabili câştigătorul. **`Bidder`**-ul va aştepta până se primeşte răspunsul în acest topic.

- **<u>Auctioneer.py</u>**

```
from kafka import KafkaConsumer, KafkaProducer


class Auctioneer:
    def __init__(self, bids_topic, notify_message_processor_topic):
        super().__init__()
        self.bids_topic = bids_topic
        self.notify_processor_topic = notify_message_processor_topic

        # consumatorul pentru ofertele de la licitatie
        self.bids_consumer = KafkaConsumer(
            self.bids_topic,
            auto_offset_reset="earliest",  # mesajele se preiau de la
cel mai vechi la cel mai recent
            group_id="auctioneers",
            consumer_timeout_ms=15_000  # timeout de 15 secunde
        )

        # producatorul pentru notificarea procesorului de mesaje
        self.notify_processor_producer = KafkaProducer()

    def receive_bids(self):
        # se preiau toate ofertele din topicul bids_topic
        print("Astept oferte pentru licitatie...")
        for msg in self.bids_consumer:
            for header in msg.headers:
                if header[0] == "identity":
                    identity = str(header[1], encoding="utf-8")
                elif header[0] == "amount":
                    bid_amount = int.from_bytes(header[1], 'big')

            print("{} a licitat {}".format(identity, bid_amount))

        # bids_consumer genereaza exceptia StopIteration atunci cand
se atinge timeout-ul de 10 secunde
        # => licitatia se incheie dupa ce timp de 15 secunde nu s-a
primit nicio oferta
        self.finish_auction()

    def finish_auction(self):
```

```
        print("Licitatia s-a incheiat!")
        self.bids_consumer.close()

        # se notifica MessageProcessor ca poate incepe procesarea
mesajelor
        auction_finished_message = bytearray("incheiat",
encoding="utf-8")

self.notify_processor_producer.send(topic=self.notify_processor_topic,
value=auction_finished_message)
        self.notify_processor_producer.flush()
        self.notify_processor_producer.close()

    def run(self):
        self.receive_bids()


if __name__ == '__main__':
    auctioneer = Auctioneer(
        bids_topic="topic_oferte",

notify_message_processor_topic="topic_notificare_procesor_mesaje"
    )
    auctioneer.run()
```

Microserviciul **`Auctioneer`** foloseşte 2 *topic*-uri Kafka, astfel:

- **`topic_oferte`** → de aici consumă mesajele cu oferte primite de la microserviciile **`Bidder`**. Se observă că fiecare instanţă de **`Auctioneer`** este asignată ca şi consumator în grupul de consumatori „**`auctioneers`**” - a se vedea parametrul **`group_id="auctioneers"`** adăugat la construirea obiectului **`KafkaConsumer`**. *Topic*-ul este împărţit în 4 partiţii, şi deci Kafka va distribui mesajele în mod echilibrat între consumatorii acelui grup. Dacă se pornesc mai multe instanţe de **`Auctioneer`**, fiecare va primi o parte din fluxul de mesaje de la entităţile **`Bidder`**.
- **`topic_notificare_procesor_mesaje`** → deoarece licitaţia se încheie după 15 secunde de inactivitate la aşteptarea de oferte (conform parametrului **`consumer_timeout_ms=15_000`**), **`MessageProcessor`** este notificat prin acest *topic* atunci când nu se mai primesc oferte, ca el să poată prelua mesajele spre procesare.

- **<u>MessageProcessor.py</u>**

```
from datetime import datetime
from kafka import KafkaConsumer, KafkaProducer


class MessageProcessor:
    def __init__(self, bids_topic, notify_message_processor_topic,
processed_bids_topic):
        super().__init__()
        self.bids_topic = bids_topic
        self.notify_message_processor_topic =
notify_message_processor_topic
        self.processed_bids_topic = processed_bids_topic

        # consumatorul notificarii de la Auctioneer cum ca s-a
```

```
terminat licitatia
        self.notify_message_processor_consumer = KafkaConsumer(
            self.notify_message_processor_topic,
            auto_offset_reset="earliest"  # mesajele se preiau de la
cel mai vechi la cel mai recent
        )

        # consumatorul pentru ofertele de la licitatie
        self.bids_consumer = KafkaConsumer(
            self.bids_topic,
            auto_offset_reset="earliest",
            consumer_timeout_ms=1000
        )

        # producatorul pentru mesajele procesate
        self.processed_bids_producer = KafkaProducer()

        # ofertele se pun in dictionar, sub forma de perechi
<IDENTITATE_OFERTANT, MESAJ_OFERTA>
        self.bids = dict()

    def get_and_process_messages(self):
        # se asteapta notificarea de la Auctioneer pentru incheierea
licitatiei
        print("Astept notificare de la toate entitatile Auctioneer
pentru incheierea licitatiei...")
        auction_end_message =
next(self.notify_message_processor_consumer)

        # a ajuns prima notificare, se asteapta si celelalte
notificari timp de maxim 15 secunde

self.notify_message_processor_consumer.config["consumer_timeout_ms"] =
15_000
        for auction_end_message in
self.notify_message_processor_consumer:
            pass
        self.notify_message_processor_consumer.close()

        if str(auction_end_message.value, encoding="utf-8") ==
"incheiat":
            # se preiau toate ofertele din topicul bids_topic si se
proceseaza
            print("Licitatie incheiata. Procesez mesajele cu
oferte...")

            for msg in self.bids_consumer:
                for header in msg.headers:
                    if header[0] == "identity":
                        identity = str(header[1], encoding="utf-8")

                # eliminare duplicate
                if identity not in self.bids.keys():
                    self.bids[identity] = msg

            self.bids_consumer.close()
```

```
            # sortare dupa timestamp
            sorted_bids = sorted(self.bids.values(), key=lambda bid:
bid.timestamp)

            self.finish_processing(sorted_bids)

    def finish_processing(self, sorted_bids):
        print("Procesarea s-a incheiat! Trimit urmatoarele oferte:")
        for bid in sorted_bids:
            for header in bid.headers:
                if header[0] == "identity":
                    identity = str(header[1], encoding="utf-8")
                elif header[0] == "amount":
                    bid_amount = int.from_bytes(header[1], 'big')
            print("[{}] {} a licitat
{}.".format(datetime.fromtimestamp(bid.timestamp / 1000), identity,
bid_amount))

            # se stocheaza mesajele ordonate dupa timestamp si fara
duplicate intr-un topic separat

self.processed_bids_producer.send(topic=self.processed_bids_topic,
value=bid.value, headers=bid.headers)

        self.processed_bids_producer.flush()
        self.processed_bids_producer.close()

    def run(self):
        self.get_and_process_messages()


if __name__ == '__main__':
    message_processor = MessageProcessor(
        bids_topic="topic_oferte",

notify_message_processor_topic="topic_notificare_procesor_mesaje",
        processed_bids_topic="topic_oferte_procesate"
    )
    message_processor.run()
```

MessageProcessor foloseşte 3 *topic*-uri Kafka:

- **topic_oferte** → de aici îşi preia mesajele pentru procesare, după ce a fost notificat că licitaţia a fost încheiată

- **topic_notificare_procesor_mesaje** → aşteaptă un mesaj trimis de **Auctioneer**, cu textul „**incheiat**” pentru a şti că nu mai sunt entităţi **Bidder** care vor mai trimite oferte, şi poate începe deci procesarea mesajelor din *topic*-ul **topic_oferte**.

- **topic_oferte_procesate** → în acest *topic* sunt produse mesajele ce conţin ofertele finale, ordonate după dată şi oră, şi fără duplicate. **Kafka asigură primirea în ordine a mesajelor din aceeaşi partiţie a unui *topic***. Deoarece **topic_oferte_procesate** este creat cu o singură partiţie, este asigurată astfel ordonarea mesajelor obţinute după procesare.

- **BiddingProcessor.py**

```
from kafka import KafkaConsumer, KafkaProducer


class BiddingProcessor:
    def __init__(self, processed_bids_topic, result_topic):
        super().__init__()
        self.processed_bids_topic = processed_bids_topic
        self.result_topic = result_topic

        # consumatorul pentru ofertele procesate
        self.processed_bids_consumer = KafkaConsumer(
            self.processed_bids_topic,
            auto_offset_reset="earliest",  # mesajele se preiau de la
cel mai vechi la cel mai recent
            consumer_timeout_ms=3000
        )

        # producatorul pentru trimiterea rezultatului licitatiei
        self.result_producer = KafkaProducer()

    def get_processed_bids(self):
        # se preiau toate ofertele procesate din topicul
processed_bids_topic
        print("Astept ofertele procesate de MessageProcessor...")

        # ofertele se stocheaza sub forma de perechi <PRET_LICITAT,
MESAJ_OFERTA>
        bids = dict()
        no_bids_available = True

        while no_bids_available:
            for msg in self.processed_bids_consumer:
                for header in msg.headers:
                    if header[0] == "amount":
                        bid_amount = int.from_bytes(header[1], 'big')
                bids[bid_amount] = msg

            # daca inca nu exista oferte, se asteapta in continuare
            if len(bids) != 0:
                no_bids_available = False


        self.processed_bids_consumer.close()
        self.decide_auction_winner(bids)

    def decide_auction_winner(self, bids):
        print("Procesez ofertele...")

        if len(bids) == 0:
            print("Nu exista nicio oferta de procesat.")
            return

        # sortare dupa oferte, descrescator
        sorted_bids = sorted(bids.keys(), reverse=True)
```

```
        # castigatorul este ofertantul care a oferit pretul cel mai
mare
        winner = bids[sorted_bids[0]]

        for header in winner.headers:
            if header[0] == "identity":
                winner_identity = str(header[1], encoding="utf-8")

        print("Castigatorul este:")
        print("\t{} - pret licitat: {}".format(winner_identity,
sorted_bids[0]))

        # se trimite rezultatul licitatiei pentru ca entitatile Bidder
sa il preia din topicul corespunzator
        self.result_producer.send(topic=self.result_topic,
value=winner.value, headers=winner.headers)
        self.result_producer.flush()
        self.result_producer.close()

    def run(self):
        self.get_processed_bids()


if __name__ == '__main__':
    bidding_processor = BiddingProcessor(
        processed_bids_topic="topic_oferte_procesate",
        result_topic="topic_rezultat"
    )
    bidding_processor.run()
```

BiddingProcessor foloseşte 2 *topic*-uri Kafka, astfel:

- **topic_oferte_procesate** → se aşteaptă primirea ofertelor procesate de la **MessageProcessor**, iar pe măsură ce acestea sunt disponibile în topic, sunt preluate şi adăugate într-o colecţie internă.
- **topic_rezultat** → după decizia câştigătorului, mesajul efectiv trimis de acesta este publicat în **topic_rezultat**, pentru ca fiecare **Bidder** să îl preia de acolo şi să afle cine a câştigat licitaţia.

Încapsularea microserviciilor în imagini Docker

Se oferă un exemplu de încapsulare a microserviciului **Auctioneer** într-o imagine Docker, pe baza căreia se poate porni un număr variabil de containere.

Creaţi un folder denumit, spre exemplu, **Auctioneer_Docker**, în care adăugaţi fişierul sursă al microserviciului **Auctioneer** (**Auctioneer.py**).

Alături de acel fişier sursă, creaţi un fişier **Dockerfile**, în care adăugaţi următorul conţinut:

```
FROM python:alpine

# instalare modul kafka-python
RUN pip install kafka-python
```

```
# adaugare fisier sursa pentru microserviciul Auctioneer
WORKDIR /auctioneer
ADD Auctioneer.py $WORKDIR

# comanda de executie este: python <nume_fisier>.py
CMD ["python", "Auctioneer.py"]
```

Fişierul Dockerfile va fi folosit pentru construirea unei imagini Docker care încapsulează microserviciul **Auctioneer**. Nu este nevoie să o construiţi imediat, deoarece acest lucru va fi făcut automat de **Docker Compose** (urmează mai jos).

Execuţia aplicaţiei

Se recomandă execuţia de la terminal, aşa încât deschideţi unul în folder-ul proiectului şi creaţi un mediu virtual Python în care instalaţi modulul **kafka-python**:

```
python3 -m venv okazii_env
source ./okazii_env/bin/activate
pip3 install kafka-python
```

Nu uitaţi să verificaţi că server-ul Kafka şi containerul Docker cu ETCD sunt pornite!

Pregătirea topic-urilor

Înainte de fiecare pornire a aplicaţiei, folosiţi următorul script Python pentru a pregăti *topic*-urile Kafka pentru pornirea licitaţiei:

- **PrepareAuction.py**

```
from kafka import KafkaAdminClient
from kafka import KafkaConsumer
from kafka.admin import NewTopic
import time

if __name__ == '__main__':
    admin = KafkaAdminClient()

    used_topics = (
        "topic_oferte",
        "topic_rezultat",
        "topic_oferte_procesate",
        "topic_notificare_procesor_mesaje",
    )

    # se sterg topic-urile, daca exista deja
    print("Se sterg topic-urile existente...")

    kafka_topics = KafkaConsumer().topics()
    for topic in kafka_topics:
        if topic in used_topics:
            print("\tSe sterge {}...".format(topic))
            admin.delete_topics(topics=[topic], timeout_ms=2000)

            # se asteapta putin ca stergerea sa aiba loc
            time.sleep(2)
```

```
    # se creeaza topic-urile necesare aplicatiei
    print("Se creeaza topic-urile necesare:")
    lista_topicuri = [
        NewTopic(name=used_topics[0], num_partitions=4,
replication_factor=1),
        NewTopic(name=used_topics[1], num_partitions=1,
replication_factor=1),
        NewTopic(name=used_topics[2], num_partitions=1,
replication_factor=1),
        NewTopic(name=used_topics[3], num_partitions=1,
replication_factor=1)
    ]
    for topic in lista_topicuri:
        print("\t{}".format(topic.name))
    admin.create_topics(lista_topicuri, timeout_ms=3000)

    print("Gata! Microserviciile participante la licitatie pot fi
pornite.")
```

Atenţie: înainte de a-l executa, asiguraţi-vă că nu există niciun consumator Kafka pornit (niciun microserviciu în aşteptare de mesaje, niciun script ce foloseşte API-ul Kafka pentru consumarea mesajelor din *topic*-urile folosite de aplicaţie).

Script-ul va şterge *topic*-urile existente pentru a forţa golirea mesajelor existente de la execuţii anterioare ale aplicaţiei. Apoi, *topic*-urile vor fi recreate cu parametrii necesari.

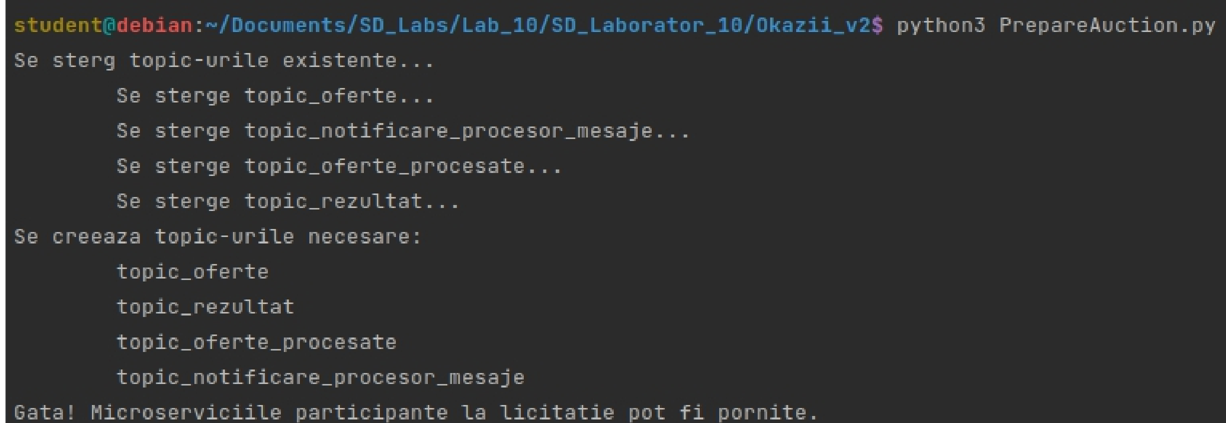


Figura 13 - Pregătire *topic*-uri Kafka

Dacă primiţi o eroare cum că există deja un anumit *topic*: măriţi timpul de aşteptare pentru ştergere (parametrul `time.sleep(...)`). Verificaţi să nu fie pornit niciun consumator de mesaje din acele *topic*-uri conţinute în tupla `used_topics`.

Dacă nu rulaţi script-ul înainte de aplicaţie, microserviciile vor reciti mesajele anterioare, deoarece sunt persistente până la expirare. Puteţi profita de această proprietate a Kafka pentru testarea independentă, izolată a microserviciilor: de exemplu, dacă porniţi doar `MessageProcessor` şi există mesaje din execuţii anterioare, acesta va reciti ofertele existente în Kafka şi le va procesa din nou, trimiţând un alt set de mesaje procesate pentru `BiddingProcessor`.

Pornirea microserviciilor

Deoarece prin varianta de implementare cu Kafka s-a obţinut un grad mare de decuplare, iar fiecare microserviciu aşteaptă mesajele necesare lui pentru a funcţiona, **microserviciile se**

pot porni în orice ordine se doreşte, cu menţiunea că **fiecare instanţă a microserviciului `Auctioneer` aşteaptă maxim 15 secunde de inactivitate din partea `Bidder`-ilor**.

Deci, se recomandă să porniţi **`Auctioneer`** ca ultimul microserviciu.

Pornirea microserviciilor care nu sunt încapsulate în imagini Docker

Deschideţi câte un terminal în folder-ul cu fişierele sursă ale proiectului, şi porniţi fiecare microserviciu (**în afară de `Auctioneer`**) astfel:

```
source CALE_CĂTRE_VENV/bin/activate
python3 NUME_MICROSERVICIU.py
```

Pornirea Auctioneer cu factor de scalare parametrizat

Microserviciul **`Auctioneer`** se porneşte folosind **Docker Compose**, pentru a putea fi scalat cu uşurinţă: astfel, se pot porni oricâte instanţe **`Auctioneer`** doreşte utilizatorul.

Pentru instalare docker-compose urmați pașii din documentație https://docs.docker.com/compose/install/ .

Creaţi un fişier numit **`docker-compose.yml`** în folder-ul **`Auctioneer_Docker`** creat anterior, şi adăugaţi următorul conţinut:

```
version: "3.7"
services:
  auctioneer:
    build:
      context: .
    network_mode: host
```

Deci, pentru a porni **2 instanţe** ale microserviciului **`Auctioneer`**, executaţi următoarea comandă în folder-ul **`Auctioneer_Docker`**, în care se regăseşte fişierul **`docker-compose.yml`**:

```
docker-compose up --scale auctioneer=2
```

Un exemplu de execuţie se regăseşte în figura următoare:

Figura 14 - Exemplu de execuţie

Puteţi observa împărţirea mesajelor pe partiţii în cazul entităţilor **`Auctioneer`** şi, de asemenea, existenţa unui mesaj duplicat care a fost filtrat de **`MessageProcessor`**.

Utilizarea Kafka pentru monitorizarea aplicaţiilor în dev-ops

Kafka poate fi utilizat şi pentru statistici şi monitorizare, de exemplu în sensul de colectare a informaţiilor din *topic*-urile utilizate de aplicaţia bazată pe microservicii: numărul de mesaje primite, numărul de mesaje procesate etc. - toate acestea pot oferi informaţii legate de **gradul de încărcare al fluxului de mesaje**.

De asemenea, se pot prelua şi metrici oferite de Docker Engine în scopul monitorizării gradului de încărcare general al aplicaţiei în timp real: **utilizare CPU**, **utilizare memorie** etc.

În scopul monitorizării aplicaţiei **Okazii** din laborator, creaţi o aplicaţie separată Spring Boot (numită, de exemplu, **`MonitoringApplication`**), la care adăugaţi dependenţa **Spring-Kafka** (ca în exemplul 2 din laborator). Această aplicaţie reprezintă un microserviciu utilizat pentru instrumentaţie şi monitorizare.

- **KafkaMonitor.kt**

```
package com.sd.laborator

import org.apache.kafka.clients.consumer.ConsumerRecord
import org.springframework.kafka.annotation.KafkaListener
import org.springframework.kafka.annotation.PartitionOffset
import org.springframework.kafka.annotation.TopicPartition
import org.springframework.scheduling.annotation.Scheduled
import org.springframework.stereotype.Component
import java.io.BufferedReader
import java.io.InputStreamReader
import java.net.URLEncoder
```

```
import java.time.LocalDateTime

@Component
class KafkaMonitor {
    companion object {
        private var numberOfBids: Int = 0
        private var numberOfProcessedBids: Int = 0

        private var BASE_DOCKER_API_COMMAND = "curl --unix-socket
/var/run/docker.sock http:/v1.40"
    }

    init {
        println("KakfaMonitor instantiated!")
    }

    @KafkaListener(
         groupId = "KafkaMonitor",
        topicPartitions = [
            TopicPartition(
                topic = "topic_oferte",
                partitionOffsets = [
                    PartitionOffset(partition = "0", initialOffset =
"0"),
                    PartitionOffset(partition = "1", initialOffset =
"0"),
                    PartitionOffset(partition = "2", initialOffset =
"0"),
                    PartitionOffset(partition = "3", initialOffset =
"0")
                ]
            ),
            TopicPartition(
                topic = "topic_oferte_procesate",
                partitionOffsets = [
                    PartitionOffset(partition = "0", initialOffset =
"0")
                ]
            )
        ]
    )
    fun monitorKafkaMessages(message: ConsumerRecord<String, String>)
{
        when(message.topic()) {
            "topic_oferte" -> ++numberOfBids
            "topic_oferte_procesate" -> ++numberOfProcessedBids
        }
    }

    @Scheduled(fixedDelay=1000)
    fun showKafkaStats() {
        println("[${LocalDateTime.now()}] Grad incarcare Auctioneer:
$numberOfBids oferte primite")
        println("[${LocalDateTime.now()}] Grad incarcare
MessageProcessor: $numberOfProcessedBids oferte procesate")
    }
```

```
    @Scheduled(fixedDelay=2000)
    fun monitorAuctioneerContainer() {
        // preluare lista de instante ale Auctioneer
        val auctioneerContainersFilter =
URLEncoder.encode("{\"ancestor\":
[\"auctioneer_docker_auctioneer:latest\"]}", "utf-8")
        val auctioneerListProcess: Process =

Runtime.getRuntime().exec("$BASE_DOCKER_API_COMMAND/containers/json?fi
lters=$auctioneerContainersFilter")
        val auctioneerListProcessInput =
BufferedReader(InputStreamReader(auctioneerListProcess.inputStream))

        // se citeste raspunsul de la Docker Engine API sub forma de
JSON
        val auctioneerListOutput =
auctioneerListProcessInput.readLine()
        auctioneerListProcessInput.close()

        // se preia lista de ID-uri ale container-elor de tip
Auctioneer
        val containerIdRegex = Regex("\"Id\":\"([a-f0-9]*)\"")
        containerIdRegex.findAll(auctioneerListOutput).forEach {
            // pentru fiecare ID in parte se preiau statisticile la
runtime
            val auctioneerContainerID = it.groupValues[1].take(12)

            // se foloseste ruta /containers/ID/stats de la API-ul
Docker Engine
            val auctioneerContainerStatsProcess: Process =

Runtime.getRuntime().exec("$BASE_DOCKER_API_COMMAND/containers/$auctio
neerContainerID/stats?stream=0")
            val auctioneerContainerStatsProcessInput =
BufferedReader(InputStreamReader(auctioneerContainerStatsProcess.input
Stream))

            // din output-ul cu statistici, se colecteaza utilizarea
CPU si a memoriei
            val auctioneerContainerStatsOutput =
auctioneerContainerStatsProcessInput.readLine()

            val cpuUsageRegex =
Regex("\"cpu_stats\":\\{\"cpu_usage\":\\{\"total_usage\":([0-9]*),")

cpuUsageRegex.find(auctioneerContainerStatsOutput)?.groupValues?.get(1)
.let {
                println("[$auctioneerContainerID] Utilizare procesor:
$it")
            }

            val memoryUsageRegex =
Regex("\"memory_stats\":\\{\"usage\":([0-9]*),")

memoryUsageRegex.find(auctioneerContainerStatsOutput)?.groupValues?.ge
```

```
t(1).let {
                println("[$auctioneerContainerID] Utilizare memorie:
$it")
            }
        }
    }
}
```

Componenta de monitorizare conţine:

– un *listener* pentru Kafka, care urmăreşte mesajele primite în *topic*-urile **`topic_oferte`**, respectiv **`topic_oferte_procesate`**, deoarece acestea ar fi punctele de interes cu posibilitate mare de încărcare. Pentru fiecare *topic* în parte, trebuie specificate partiţiile care vor fi urmărite, respectiv *offset*-ul de la care se începe citirea mesajelor (**0** = de la început). Fiind aplicaţie de monitorizare, toate cele 4 partiţii de la *topic*-ul **`topic_oferte`** sunt de interes. *Listener*-ul contorizează mesajele primite, în timp real.

– o funcţie recurentă care afişează numărul de mesaje preluate de *listener*-ul Kafka: **`showKafkaStats()`**

– o funcţie recurentă care monitorizează instanţele microserviciului **`Auctioneer`** sub formă de containere Docker (indiferent de numărul lor, în mod dinamic): **`showAuctioneerContainersStats()`**. Funcţia utilizează API-ul Docker Engine (https://docs.docker.com/engine/api/v1.40) pentru a trimite cereri către procesul *daemon* al Docker prin socket-ul dedicat:

  – **`/containers/json`** → afişează informaţii despre containerele în execuţie. Se pot filtra rezultatele cu parametrul **`filters`**.

  – **`/containers/<ID_CONTAINER>/stats`** → afişează metrici despre un anumit container. Pentru a primi doar ultimele valori (şi nu un flux continuu), se foloseşte parametrul URL **`stream=0`**.

Metodele adnotate cu **`@Scheduled`** se execută o dată la perioada de timp specificată ca parametru (în milisecunde).

- **MonitoringApplication.kt**

```
package com.sd.laborator

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication
import org.springframework.scheduling.annotation.EnableScheduling

@SpringBootApplication
@EnableScheduling
class MonitoringApplication

fun main(args: Array<String>) {
    runApplication<MonitoringApplication>(*args)
}
```

S-a folosit adnotarea **`@EnableScheduling`** pentru a activa posibilitatea de a folosi funcţiile recurente din Spring (adnotate cu **`@Scheduler`**).

Utilizaţi aplicaţia de monitorizare pentru a prelua metrici în timp real din aplicaţia Okazii.

Porniţi **MonitoringApplication** după ce a început licitaţia, şi veţi putea găsi metricile afişate în *log*-urile Spring:

```
[2021-11-28T14:03:52.752] Grad incarcare MessageProcessor: 3 oferte procesate
[2021-11-28T14:03:53.758] Grad incarcare Auctioneer: 5 oferte primite
[2021-11-28T14:03:53.758] Grad incarcare MessageProcessor: 3 oferte procesate
[2021-11-28T14:03:54.759] Grad incarcare Auctioneer: 5 oferte primite
[2021-11-28T14:03:54.759] Grad incarcare MessageProcessor: 3 oferte procesate
[509ae8206c08] Utilizare procesor: 131479000
[509ae8206c08] Utilizare memorie: 11485184
[063dded61356] Utilizare procesor: 141823000
[063dded61356] Utilizare memorie: 11591680
[2021-11-28T14:03:58.789] Grad incarcare Auctioneer: 5 oferte primite
[2021-11-28T14:03:58.789] Grad incarcare MessageProcessor: 3 oferte procesate
[2021-11-28T14:03:59.789] Grad incarcare Auctioneer: 6 oferte primite
[2021-11-28T14:03:59.789] Grad incarcare MessageProcessor: 3 oferte procesate
[509ae8206c08] Utilizare procesor: 152176000
[509ae8206c08] Utilizare memorie: 11579392
[063dded61356] Utilizare procesor: 165319000
[063dded61356] Utilizare memorie: 11612160
[2021-11-28T14:04:03.817] Grad incarcare Auctioneer: 9 oferte primite
[2021-11-28T14:04:03.817] Grad incarcare MessageProcessor: 7 oferte procesate
[2021-11-28T14:04:04.818] Grad incarcare Auctioneer: 9 oferte primite
[2021-11-28T14:04:04.818] Grad incarcare MessageProcessor: 7 oferte procesate
[509ae8206c08] Utilizare procesor: 187469000
[509ae8206c08] Utilizare memorie: 11595776
[063dded61356] Utilizare procesor: 187867000
[063dded61356] Utilizare memorie: 11616256
[2021-11-28T14:04:08.854] Grad incarcare Auctioneer: 9 oferte primite
[2021-11-28T14:04:08.854] Grad incarcare MessageProcessor: 7 oferte procesate
[2021-11-28T14:04:09.854] Grad incarcare Auctioneer: 9 oferte primite
[2021-11-28T14:04:09.854] Grad incarcare MessageProcessor: 7 oferte procesate
```

Figura 15 - Monitorizarea aplicaţiei Okazii

Valorile pentru utilizarea procesorului şi a memoriei sunt de fapt valori brute raportate de sistemul de fişiere virtual **/proc/stats**. Pentru a calcula efectiv procentele de utilizare, se poate face un calcul suplimentar în maniera explicată aici:
https://rosettacode.org/wiki/Linux_CPU_utilization

Aplicaţii şi teme

Teme de laborator

1. Încapsulaţi şi microserviciul **Bidder** într-o imagine Docker. Creaţi Dockerfile pentru acesta şi folosiţi comanda **docker build** (exemplu de folosire în **laboratorul 8**) pentru construirea imaginii.
2. Utilizând **Docker Compose**, porniţi un număr mare de astfel de microservicii (de exemplu, 100) la o execuţie a aplicaţiei. Scalaţi microserviciul **Auctioneer** la un număr de 4 instanţe şi verificaţi că aplicaţia face faţă fluxului mare de mesaje.

Teme pentru acasă

1. Completaţi aplicaţia de monitorizare a resurselor astfel încât să afişeze un grafic în timp

real pentru resursele monitorizate.
Puteţi folosi, eventual, biblioteca **Plotly**:

- https://github.com/mipt-npm/plotly.kt
- https://medium.com/@altavir/plotly-kt-a-dynamic-approach-to-visualization-in-kotlin-38e4feaf61f7

2. Creați câte o schemă de microservicii care ar implementa separat licitația engleza, cu lumânarea, olandeză și suedeză. În proiect trebuie prezentate aceste diagrame.

a) Apoi faceți o analiză și vedeți ce uservicii sunt similare și care sunt diferite (faceți un tabel centralizator.

b) Apoi propuneți o nouă diagrama de servicii pentru un sistem care le-ar implementa pe toate și ar putea alege ce tip de licitație să se efectueze. Atenție trebuie analizat cazul în care am mai mulți utilizatori și fiecare va alege alt tip de licitație. Acest lucru va conduce la modificări în structura userviciilor.

c) Realizați verificarea SOLID pentru uservicii si eventual modificați prima propunere

d) realizați diagrama UML

e) Implementați și testați sistemul rezultat

Găsiţi regulile licitaţiei engleze şi nu numai aici:
https://corporatefinanceinstitute.com/resources/knowledge/finance/english-auction/

Bibliografie

[1]: Neha Narkhede, Kafka: The Definitive Guide - Real-Time Data and Stream Processing at Scale
[2]: Introducere în Apache Kafka - https://kafka.apache.org/intro
[3]: Ghid rapid pentru instalare şi configurare Kafka - https://kafka.apache.org/quickstart
[4]: ETCD - https://etcd.io/
[5]: Biblioteca Kafka-Python - https://github.com/dpkp/kafka-python
[6]: API-ul Kafka-Python - https://kafka-python.readthedocs.io/en/master/apidoc/modules.html
[7]: Proiectul Spring Kafka - https://spring.io/projects/spring-kafka
[8]: Documentaţie Spring Kafka (**codul din documentaţie este scris în Java - se poate transforma / adapta pentru limbajul Kotlin**) - https://docs.spring.io/spring-kafka/docs/2.4.5.RELEASE/reference/html/
[9] Introducere în Docker Compose - https://docs.docker.com/compose/gettingstarted/
[10] Docker Compose - formatul de fişier YAML pentru configurare - https://docs.docker.com/compose/compose-file/
[11] Scalarea containerelor folosind Docker Compose - https://docs.docker.com/compose/reference/scale/
[12] Monitorizarea containerelor Docker - comanda **`docker stats`** - https://docs.docker.com/engine/reference/commandline/stats/
[13] API-ul Docker Engine - https://docs.docker.com/engine/api/v1.40/
[14] Exemple de utilizare ale API-ului Docker Engine - https://docs.docker.com/engine/api/sdk/examples/