

Sisteme Distribuite - Laborator 10

Aplicații flux și 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”. Față 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 și în ordinea în care au fost scrise.

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

Mesajele în Kafka sunt organizate pe **subiecte** (eng. *topics*). Acestea pot fi asemănat cu un folder în sistemul de fișiere sau un tabel într-o bază de date. La rândul lor, subiectele sunt împărțite în **partiții** (eng. *partitions*). O partiție este echivalentă cu un *commit log*. Partițiile reprezintă o modalitate prin care Kafka oferă redundanță și scalabilitate.

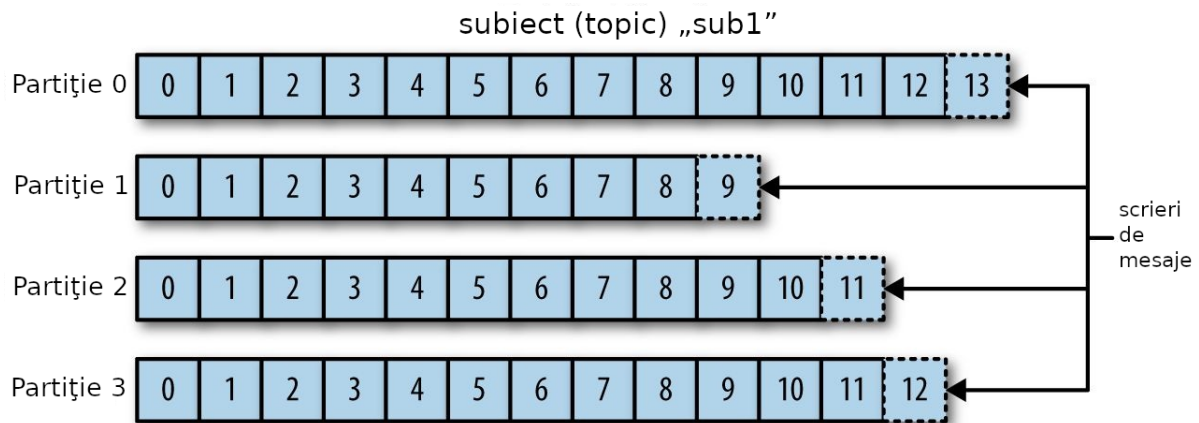


Figura 1 - Subiect Kafka cu 4 partiții

Entitățile componente utilizate în laborator, precum și relațiile dintre acestea sunt scoase în evidență în următoarea diagramă:

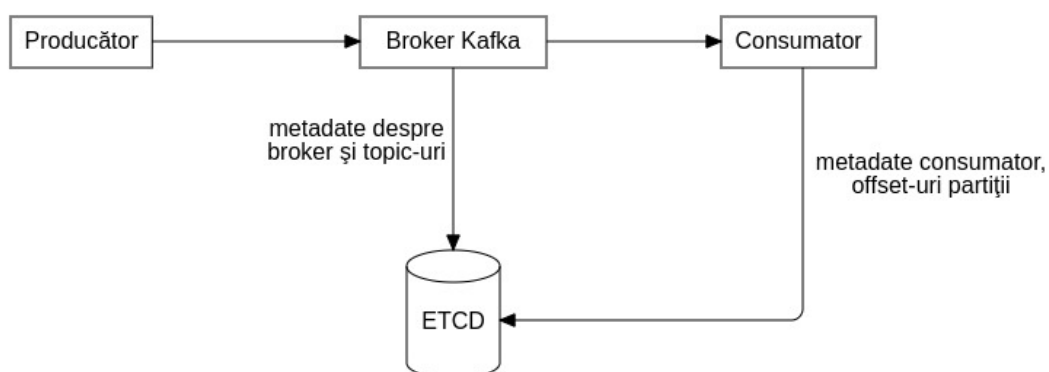


Figura 2 - Funcționarea Kafka și ETCD

ETCD

ETCD este un depozit distribuit de tip cheie-valoare, care asigură o consistență sporită și furnizează o modalitate sigură de a stoca date ce trebuie accesate de un sistem distribuit sau de un cluster de mașini de calcul. De asemenea, gestionează automat alegerile de lider din timpul partiționărilor rețelei și poate tolera defecte ale mașinilor de calcul, chiar și la nivelul nodului lider.

Datele sunt stocate în directoare organizate ierarhic, asemenea unui sistem de fișiere 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, operațiile nu actualizează structura de date, ci mereu generează o altă structură actualizată, având posibilitatea de a accesa și versiunile anterioare ale valorilor corespunzătoare cheilor.

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

Instalare și configurare Apache Kafka

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

Aveți nevoie de JRE versiunea 8 și de Docker Engine instalate pe sistemul dvs. pentru ca pașii următori să funcționeze 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, creați folder-ele în care ETCD își va stoca fișierele temporare, respectiv datele și asigurați-le 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, porniți o instanță local de ETCD, cu următoarea comandă (**dacă nu doriți repornirea automată a containerului, eliminați parametrul marcat cu roșu**):

```
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
```

Folosiți comanda **docker logs** ca să verificați funcționarea serviciului ETCD. Preluăți ID-ul container-ului Docker returnat de comanda anterioară (sau folosiți comanda **docker ps**) și executați:

```
docker logs <ID_CONTAINER_ETCD>
```

Un exemplu de output este cel din figura următoare:

```
cosmin@debian-gl553v:~$ docker logs a68338afe01c27e1d701ffc9a083ab8436113fa2212e1a58668386b899fc67f
{"level":"info","ts":"2020-04-21T15:52:15.553Z","caller":"etcdmain/etcd.go:134","msg":"server has been already initialized","data-dir":"/var/etcd-data","dir-type":"member"}
{"level":"info","ts":"2020-04-21T15:52:15.553Z","caller":"embed/etcd.go:117","msg":"configuring peer listeners","listen-peer-urls":["http://0.0.0.0:2380"]}
{"level":"info","ts":"2020-04-21T15:52:15.553Z","caller":"embed/etcd.go:127","msg":"configuring client listeners","listen-client-urls":["http://0.0.0.0:2379"]}
{"level":"info","ts":"2020-04-21T15:52:15.553Z","caller":"embed/etcd.go:299","msg":"starting an etcd server","etcd-version":"3.4.7","git-sha":"e694b7bb0","go-version":"go1.12.17","go-os":"linux","go-arch":"amd64","max-cpu-set":8,"max-cpu-available":8,"member-initialized":true,"name":"sl","data-dir":"/var/etcd-data","wal-dir":"","wal-dir-dedicated":"","member-dir":"/var/etcd-data/member","force-new-cluster":false,"heartbeat-interval":"100ms","election-timeout":"1s","initial-election-tick-advance":true,"snapshot-count":100000,"snapshot-catchup-entries":5000,"initial-advertise-peer-urls":["http://0.0.0.0:2380"],"listen-peer-urls":["http://0.0.0.0:2380"],"advertise-client-urls":["http://0.0.0.0:2379"],"listen-client-urls":["http://0.0.0.0:2379"],"cors":["*"],"host-whitelist":["*"],"initial-cluster":"","initial-cluster-state":"new","initial-cluster-token":"","quota-size-bytes":2147483648,"pre-vote":false,"initial-corrupt-check":false,"corrupt-check-time-interval":"0s","auto-compaction-mode":"periodic","auto-compaction-retention":"0s","auto-compaction-interval":"0s","discovery-url":"","discovery-proxy":""}
{"level":"info","ts":"2020-04-21T15:52:15.554Z","caller":"etcdserver/backend.go:79","msg":"opened backend db","path":"/var/etcd-data/member/snap/db","took":"522.232µs"}
{"level":"info","ts":"2020-04-21T15:52:15.557Z","caller":"etcdserver/raft.go:506","msg":"restarting local member","cluster-id":"f9f44c4ba0e96dd8","local-member-id":"59a9c584ea2c3f35","commit-index":382}
{"level":"info","ts":"2020-04-21T15:52:15.557Z","caller":"raft/raft.go:1530","msg":"59a9c584ea2c3f35 switched to configuration voters=()"}
{"level":"info","ts":"2020-04-21T15:52:15.557Z","caller":"raft/raft.go:700","msg":"59a9c584ea2c3f35 became follower at term 4"}
{"level":"info","ts":"2020-04-21T15:52:15.557Z","caller":"raft/raft.go:383","msg":"newRaft 59a9c584ea2c3f35 [peers: [], term: 4, commit: 382, applied: 0, lastindex: 382, lastterm: 4]"}
{"level":"warn","ts":"2020-04-21T15:52:15.560Z","caller":"auth/store.go:1318","msg":"simple token is not cryptographically signed"}
{"level":"info","ts":"2020-04-21T15:52:15.561Z","caller":"etcdserver/quota.go:98","msg":"enabled backend quota with default value","quota-name":"v3-applier","quota-size-bytes":2147483648,"quota-size":"2.1 GB"}
{"level":"info","ts":"2020-04-21T15:52:15.562Z","caller":"etcdserver/server.go:792","msg":"starting etcd server","local-member-id":"59a9c584ea2c3f35","local-server-version":"3.4.7","cluster-version":"to be decided"}
{"level":"info","ts":"2020-04-21T15:52:15.562Z","caller":"etcdserver/server.go:680","msg":"starting initial election tick advance","election-ticks":10}
{"level":"info","ts":"2020-04-21T15:52:15.562Z","caller":"raft/raft.go:1530","msg":"59a9c584ea2c3f35 switched to configuration voters=(6460912315094810421)"}
{"level":"info","ts":"2020-04-21T15:52:15.562Z","caller":"membership/cluster.go:392","msg":"added member","cluster-id":"f9f44c4ba0e96dd8","local-member-id":"59a9c584ea2c3f35","added-peer-id":"59a9c584ea2c3f35","added-peer-peer-urls":["http://0.0.0.0:2380"]}
{"level":"info","ts":"2020-04-21T15:52:15.562Z","caller":"membership/cluster.go:558","msg":"set initial cluster version","cluster-id":"f9f44c4ba0e96dd8","local-member-id":"59a9c584ea2c3f35","cluster-version":"3.4"}
{"level":"info","ts":"2020-04-21T15:52:15.562Z","caller":"api/capability.go:76","msg":"enabled capabilities for version","cluster-version":"3.4"}
{"level":"info","ts":"2020-04-21T15:52:15.563Z","caller":"embed/etcd.go:241","msg":"now serving peer/client/metrics","local-member-id":"59a9c584ea2c3f35","initial-advertise-peer-urls":["http://0.0.0.0:2380"],"listen-peer-urls":["http://0.0.0.0:2380"],"advertise-client-urls":["http://0.0.0.0:2379"],"listen-client-urls":["http://0.0.0.0:2379"],"listen-metrics-urls":[]}
{"level":"info","ts":"2020-04-21T15:52:15.563Z","caller":"embed/etcd.go:576","msg":"serving peer traffic","address":["::]:2380"}
{"level":"info","ts":"2020-04-21T15:52:17.458Z","caller":"raft/raft.go:923","msg":"59a9c584ea2c3f35 is starting a new election at term 4"}
{"level":"info","ts":"2020-04-21T15:52:17.458Z","caller":"raft/raft.go:713","msg":"59a9c584ea2c3f35 became candidate at term 5"}
{"level":"info","ts":"2020-04-21T15:52:17.458Z","caller":"raft/raft.go:824","msg":"59a9c584ea2c3f35 received MsgVoteResp from 59a9c584ea2c3f35 at term 5"}
{"level":"info","ts":"2020-04-21T15:52:17.458Z","caller":"raft/raft.go:765","msg":"59a9c584ea2c3f35 became leader at term 5"}
{"level":"info","ts":"2020-04-21T15:52:17.458Z","caller":"raft/node.go:325","msg":"raft.node: 59a9c584ea2c3f35 elected leader 59a9c584ea2c3f35 at term 5"}
{"level":"info","ts":"2020-04-21T15:52:17.459Z","caller":"etcdserver/server.go:2026","msg":"published local member to cluster through raft","local-member-id":"59a9c584ea2c3f35","local-member-attributes":{"Name":"sl","ClientURLs":["http://0.0.0.0:2379"]},"request-path":"/members/59a9c584ea2c3f35/attributes","cluster-id":"f9f44c4ba0e96dd8","publish-timeout":"7s"}
{"level":"info","ts":"2020-04-21T15:52:17.460Z","caller":"embed/serve.go:139","msg":"serving client traffic insecurely; this is strongly discouraged!","address":["::]:2379"}
```

Figura 3 - Verificarea funcționării serviciului ETCD containerizat

Compilare Kafka cu suport ETCD

Aveți 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
```

Se modifică fișierul **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
```

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

Instalare Apache Kafka

Folosiți versiunea modificată de Apache Kafka, compilată cu pașii de mai sus, care poate utiliza ETCD cu rol de serviciu de configurare. Se consideră că aveți arhiva **kafka_2.11-2.0.0-SNAPSHOT.tgz** obținută în urma compilării Kafka.

Dezarhivați arhiva:

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

Copiați conținutul în folder-ul **/opt**:

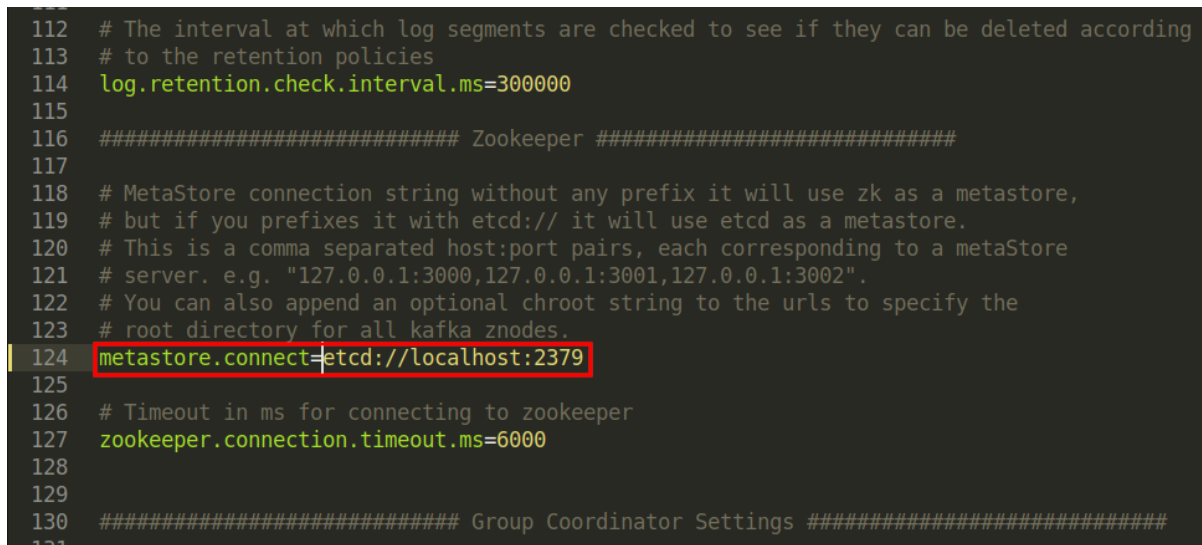
```
sudo mv kafka_2.11-2.0.0-SNAPSHOT /opt/kafka
```

Asignați-vă utilizatorul și grupa ca și proprietari ai folder-ului **kafka**:

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

Configurare Kafka pentru utilizarea ETCD ca serviciu de configurare

Deschideți fișierul **/opt/kafka/config/server.properties** cu un editor de text și modificați proprietatea **metastore.connect** astfel încât să aibă valoarea **etcd://localhost:2379**.



```
112 # The interval at which log segments are checked to see if they can be deleted according
113 # to the retention policies
114 log.retention.check.interval.ms=300000
115
116 ##### Zookeeper #####
117
118 # MetaStore connection string without any prefix it will use zk as a metastore,
119 # but if you prefixes it with etcd:// it will use etcd as a metastore.
120 # This is a comma separated host:port pairs, each corresponding to a metaStore
121 # server. e.g. "127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002".
122 # You can also append an optional chroot string to the urls to specify the
123 # root directory for all kafka znodes.
124 metastore.connect=etcd://localhost:2379
125
126 # Timeout in ms for connecting to zookeeper
127 zookeeper.connection.timeout.ms=6000
128
129
130 ##### Group Coordinator Settings #####
131
```

Figura 4 - Setarea ETCD ca serviciu de configurare Kafka

Pornire server Kafka

Nu uitați să aveți pornit containerul Docker cu ETCD înainte de a porni Kafka!

Server-ul Apache Kafka poate fi pornit astfel:

```
sudo /opt/kafka/bin/kafka-server-start.sh -daemon
/opt/kafka/config/server.properties
```

(comanda nu afișează nimic după execuție)

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 mulți parametri în linie de comandă:


```
File Edit View Terminal Tabs Help
cosmin@debian-g153v:~$ ps -ef | grep kafka
root      25752  1 54 20:26 pts/3    00:00:05 java -Xmx1G -Xms1G -server -XX:+UseG1GC -XX:MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=35 -XX:+ExplicitGCInvokesC
oncurrent -Djava.awt.headless=true -Xloggc:/opt/kafka/bin/../logs/kafkaServer-gc.log -verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+UseGC
LogFileRotation -XX:NumberOfGCLogFiles=10 -XX:GCLogFileSize=100M -Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxrem
ote.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:/o
pt/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.jar:/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/../libs/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-protobuf-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/bi
n/../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:/opt/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-base-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.jar:/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-client-2.25.1.jar:/opt/kafka/bin/../libs/jer
sey-common-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/kaf
ka/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-con
tinuation-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-secur
ity-9.2.24.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-serv
lets-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/ka
fka/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-SNAPSHOT.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-SNAPSHOT.jar:/opt/kafka/bin/../libs/kafka-streams-scala-2.11-2.0.0-SNAPSHOT.jar:/opt/kafka/bin/../libs/kafka-streams-te
st-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/kafk
a/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-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-handler-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:/o
pt/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/kaf
ka/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/snappy-java-1
.1.7.1.jar:/opt/kafka/bin/../libs/validation-api-1.1.0.Final.jar:/opt/kafka/bin/../libs/zkclient-0.10.0.jar:/opt/kafka/bin/../libs/zookeeper-3.4.12.jar kafka.Kafka /opt/kafka/
config/server.properties
cosmin  25879 30836  0 20:26 pts/3    00:00:00 grep kafka
```

Figura 5 - Verificarea funcționării server-ului Kafka

Testare broker Kafka

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

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

Verificați *topic*-ul creat astfel:

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

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

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

Comanda de mai sus va aștepta să introduceți, linie cu linie, câte un mesaj. Introduceți câte unul, după care apăsați ENTER. Când ați terminat, apăsați CTRL+D.

Testați acum consumarea mesajelor de test produse mai sus:

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

Comanda va prelua mesajele de test anterioare și le va afișa pe rând la consolă. Pentru a elibera consola și a ieși din procesul consumator de mesaje, apăsați CTRL+C.

Pentru a șterge un subiect (topic), puteți folosi comanda:

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

Atenție: ștergerea unui topic este de tip lazy. Nu are efect imediat!

Nu uitați că, atât pe parcursul laboratorului, cât și atunci când se lucrează cu aplicațiile din laborator, aveți nevoie de instanța containerizată de ETCD și de server-ul Apache Kafka pornite!

Exemplu de aplicație Kafka utilizând Python

Creați un proiect Python, iar în mediul virtual instalați modulul **kafka-python**, deoarece interfațarea 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 doriți să executați codul Python direct din PyCharm, se poate instala modulul **kafka-python astfel:** File → Settings → Project: NUME_PROIECT → Python Interpreter → apăsați pe semnul plus din dreapta (Install) și căutați „kafka-python”. Apăsați pe „Install Package” după ce ați selectat modulul din lista de rezultate.

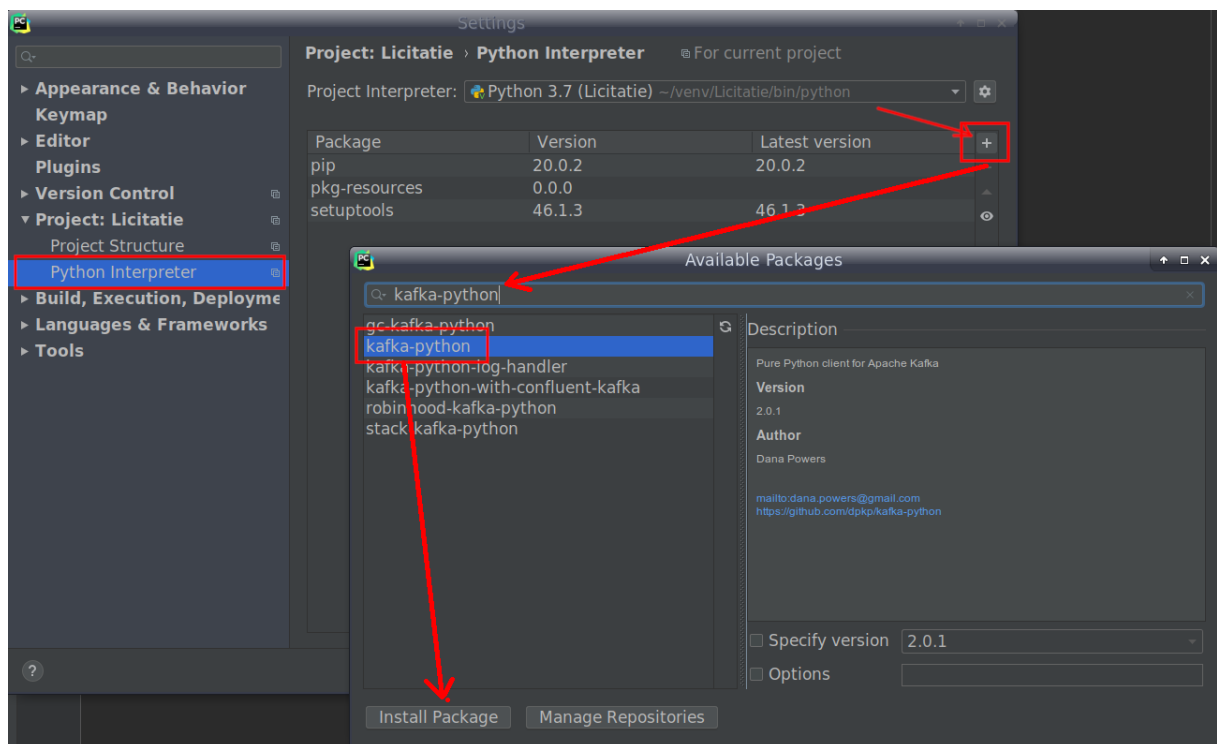


Figura 6 - Instalarea modulului **kafka-python** din PyCharm

Următorul cod exemplu folosește Kafka pentru a trimite mesaje între 2 entități: un *thread* producător de mesaje și un *thread* consumator de mesaje. Cele 2 *thread*-uri comunică prin *topic*-ul **topic_exemplu_python**. Funcționarea aplicației este redată în următoarea diagramă de activitate:

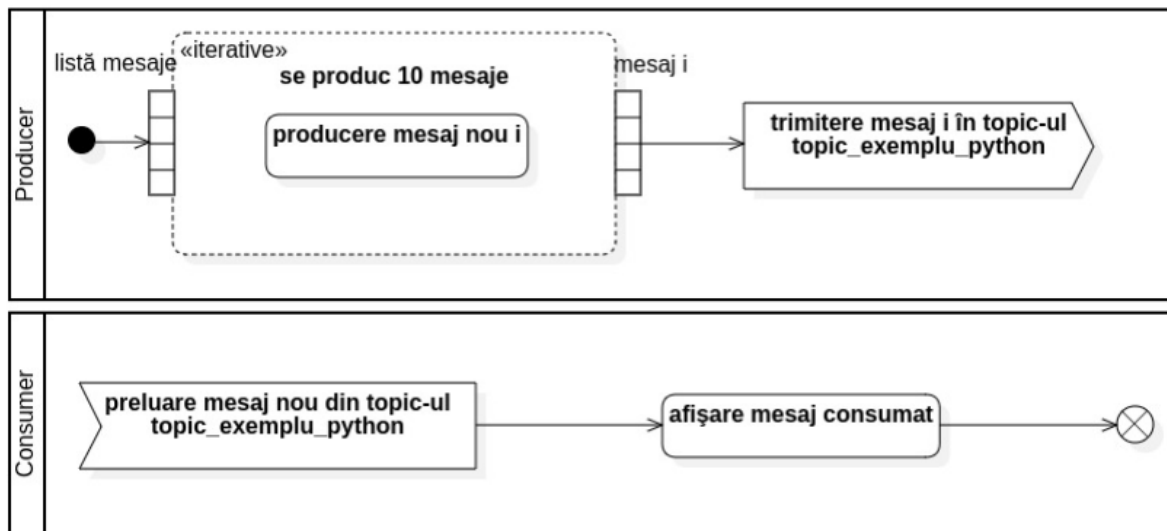


Figura 7 - Funcționarea aplicației exemplu de tip producător-consumator

• **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 primește 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()

```

Încercați să executați aplicația de 2-3 ori. Ce observați că afișează *thread*-ul consumator de mesaje? De ce apar și mesajele anterioare?

Exemplu de aplicație Kafka utilizând Kotlin

Aplicația 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 menționat apare un mesaj nou, *listener*-ul consumă respectivul mesaj și așteaptă în continuare alte mesaje.

Creați un proiect **Spring Boot** (ca în laboratorul 3), **fără să adăugați dependența Spring Web**.

Adăugați dependența **Spring Kafka** astfel:

- pentru **Maven**:

```

<dependency>
  <groupId>org.springframework.kafka</groupId>
  <artifactId>spring-kafka</artifactId>
  <version>2.2.9.RELEASE</version>
</dependency>

```

- pentru **Gradle**:

```

compile group: 'org.springframework.kafka', name: 'spring-kafka',
version: '2.2.9.RELEASE'

```

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

Atenție la versiune: 2.2.9 funcționează corect, având în vedere combinațiile de versiuni Kafka și 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)
}
```

Compilați, împachetați și executați aplicația Spring Boot.

Pentru testare, puteți 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, puteți folosi aplicația **Postman**.

Ar trebui să găsiți în log-ul aplicației Spring Boot un mesaj de la componenta **Consumer**, de tipul celui din figură:

```
2020-04-25 11:37:08.955 INFO 9508 --- [nio-8080-exec-4] o.a.kafka.common.utils.AppInfoParser : Kafka
version : 2.0.1
2020-04-25 11:37:08.955 INFO 9508 --- [nio-8080-exec-4] o.a.kafka.common.utils.AppInfoParser : Kafka
commitId : fa14705e51bd2ce5
2020-04-25 11:37:08.963 INFO 9508 --- [ad | producer-1] org.apache.kafka.clients.Metadata : Cluster
ID: TzA2yZeDR0iNMg7Pr39bSg
Am consumat urmatorul mesaj: mesaj test 1
```

Figura 8 - Testarea aplicației Kafka cu Kotlin

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

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

Proiectarea inițială a aplicației Okazii folosind Kafka pentru comunicare

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

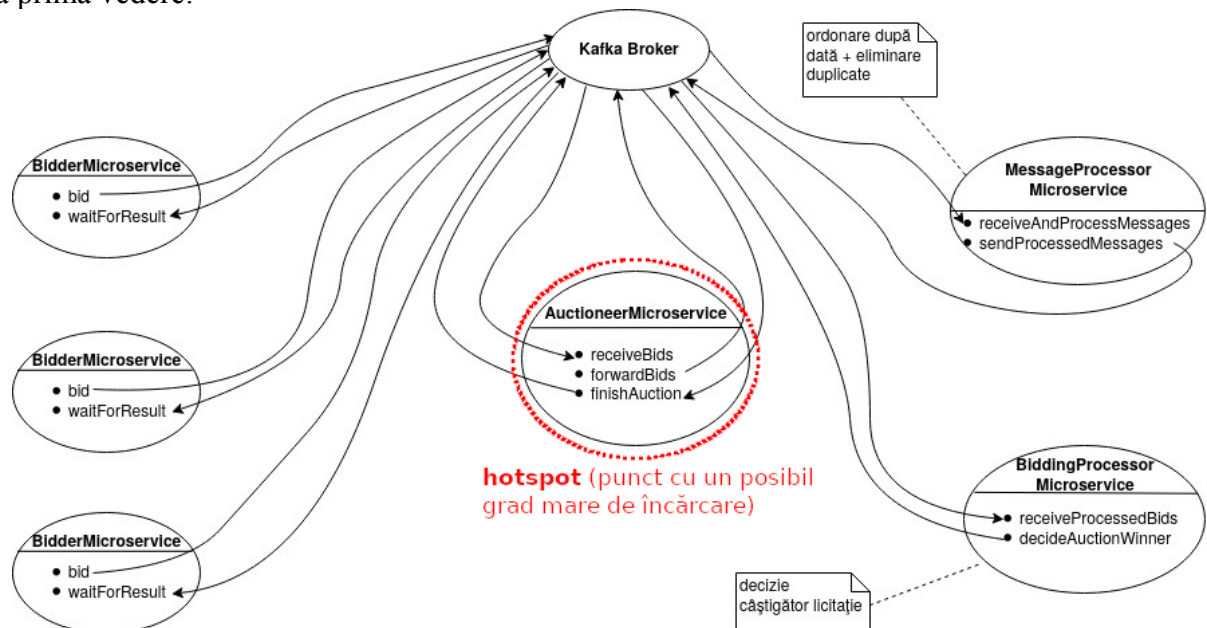


Figura 9 - Diagrama modificată de microservicii a aplicației Okazii (cu broker Kafka)

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

Se pot face însă câteva observații pentru îmbunătățirea și reproiectarea aplicației:

- Odată ce un mesaj este consumat dintr-un *topic*, el rămâne stocat până în momentul în care expiră perioada definită de **politica de retenție** (implicit, mesajele sunt șterse după 168 de ore de la publicare). Așadar, faptul că **AuctioneerMicroservice** consumă un mesaj primit de la o entitate **Bidder** nu determină distrugerea imediată a mesajului din modelul de persistență 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 retenție implicită a Kafka (/opt/kafka/config/server.properties)

- Fiecare consumator are câte un *offset* în cadrul unei partiții dintr-un *topic*. Utilizând acel *offset*, consumatorul poate citi mesajele de câte ori dorește, în ce direcție dorește. Așadar, **microservicii diferite pot citi din același topic în mod independent, fiecare având propriul offset.**

- Kafka permite ca mesajele să fie preluate simultan de mai mulți consumatori. Se pot defini astfel așa-numitele **grupuri de consumatori** (eng. *consumer groups*). Acest lucru ajută la eliminarea redundanței, în sensul că: spre deosebire de modalitatea de procesare a mesajelor din laboratorul 7, microserviciul **MessageProcessor** își poate prelua singur setul de mesaje pe care trebuie să le proceseze **direct din același topic în care le primește microserviciul Auctioneer** (consecință a observației anterioare).

- Deci, nu mai este nevoie ca microserviciul **Auctioneer** să redirecționeze 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 și 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 folosește un algoritm de tip **Round Robin** pentru a citi mesajele în mod echilibrat, în funcție de numărul de consumatori din grupă. Acest lucru asigură și **toleranța la defecte**, din punct de vedere al consumatorilor: dacă un microserviciu consumator se defectează, **mesajele vor fi redistribuite în funcție de celelalte entități active din grupul de consumatori** și de numărul de partiții în care a fost împărțit *topic*-ul care este folosit. **Mesajele nu se pierd în caz de defecte! Se redistribuie.**

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

Reproiectarea aplicației Okazii

În continuare, se reproiectează aplicația, aplicându-se proprietățile și avantajele oferite de Kafka, conform cu observațiile anterioare.

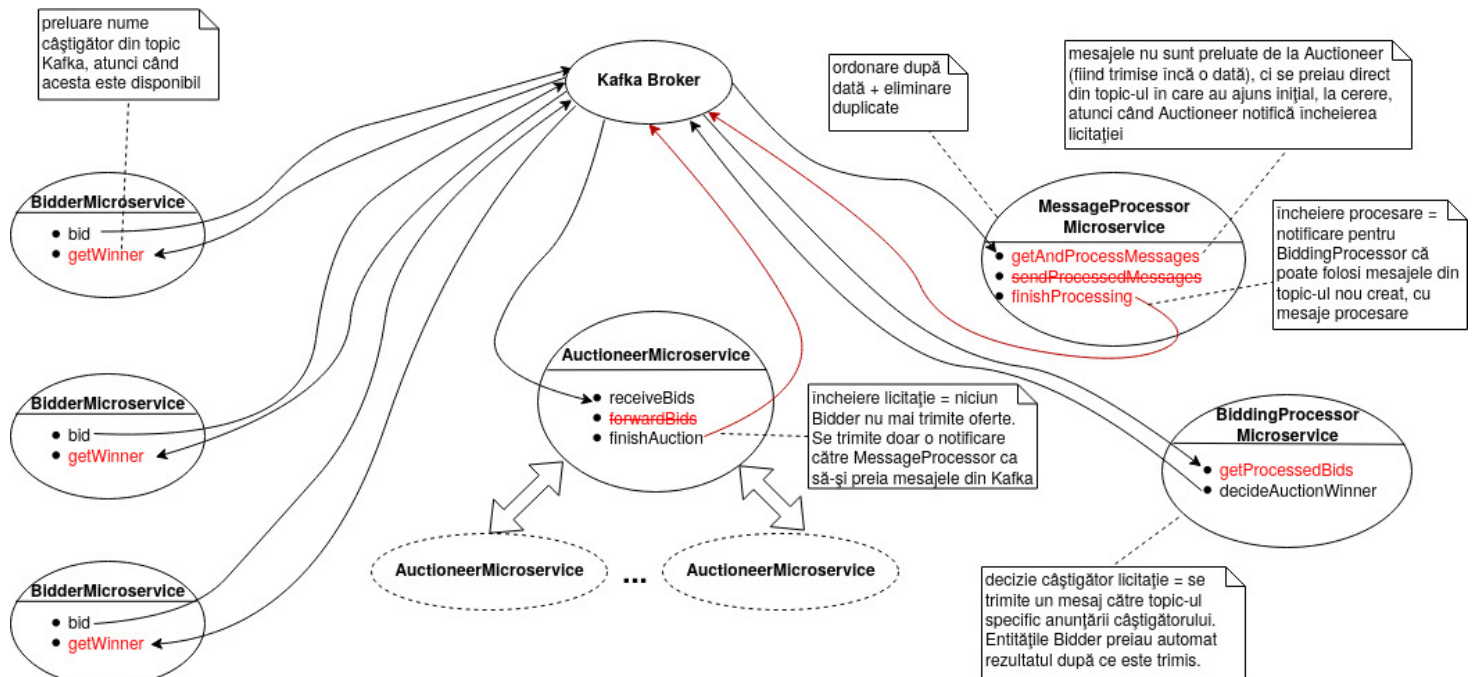


Figura 11 - Diagrama de microservicii reproiectată

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

Implementarea microserviciilor

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

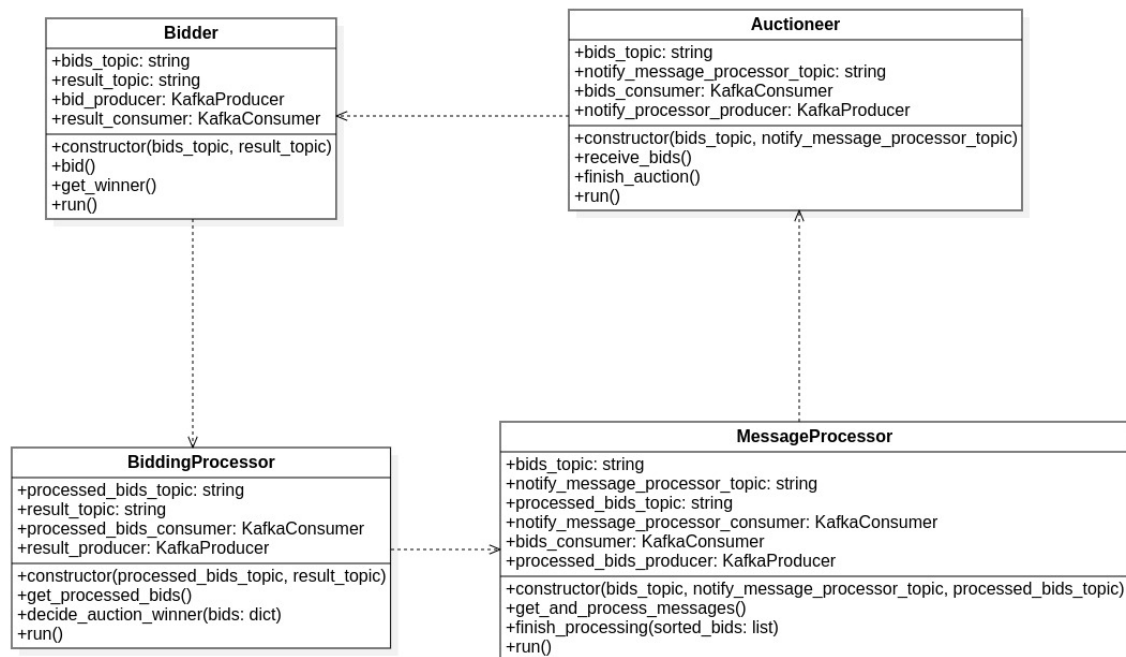


Figura 12 - Diagrama de clase aplicației Okazii

Relațiile de dependență provin de la necesitatea de comunicare prin broker-ul de mesaje. Așadar, aceste relații s-ar putea traduce prin „are nevoie să comunice cu ...”, sau „așteaptă un mesaj de la...”.

Creați un proiect Python în PyCharm și adăugați fișierele sursă menționate 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 construiesc 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 folosește de 2 *topic*-uri Kafka:

- **topic_oferte** → aici își va publica oferta (generată aleator) sub formă de mesaj Kafka. Mesajul este de forma:
 - corp: șirul de caractere „**licitez**”
 - metadata:
 - **amount**: **PRETUL_LICITAT**
 - **identity**: **IDENTIFICATOR_OFERTANT**
- **topic_rezultat** → din acest *topic* va consuma un singur mesaj: rezultatul licitației care va apărea în momentul în care **BiddingProcessor** va stabili câștigătorul. **Bidder**-ul va aștepta până se primește răspunsul în acest topic.

• Auctioneer.py

```

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** folosește 2 *topic*-uri Kafka, astfel:

- **topic_oferte** → de aici consumă mesajele cu oferte primite de la microserviciile **Bidder**. Se observă că fiecare instanță de **Auctioneer** este asignată ca și consumator în grupul de consumatori „**auctioneers**” - a se vedea parametrul **group_id="auctioneers"** adăugat la construirea obiectului **KafkaConsumer**. *Topic*-ul este împărțit în 4 partiții, și deci Kafka va distribui mesajele în mod echilibrat între consumatorii acelui grup. Dacă se pornesc mai multe instanțe de **Auctioneer**, fiecare va primi o parte din fluxul de mesaje de la entitățile **Bidder**.
- **topic_notificare_procesor_mesaje** → deoarece licitația se încheie după 15 secunde de inactivitate la așteptarea 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.

• **MessageProcessor.py**

```
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 folosește 3 *topic*-uri Kafka:

- **topic_oferte** → de aici își preia mesajele pentru procesare, după ce a fost notificat că licitația a fost încheiată
- **topic_notificare_procesor_mesaje** → așteaptă un mesaj trimis de **Auctioneer**, cu textul „încheiat” pentru a ști că nu mai sunt entități **Bidder** care vor mai trimite oferte, și poate începe deci procesarea mesajelor din *topic*-ul **topic_oferte**.
- **topic_oferte_procesate** → în acest *topic* sunt produse mesajele ce conțin ofertele finale, ordonate după dată și oră, și fără duplicate. **Kafka asigură primirea în ordine a mesajelor din aceeași partiție a unui topic**. Deoarece **topic_oferte_procesate** este creat cu o singură partiție, este asigurată astfel ordonarea mesajelor obținute 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 licitației
        self.result_producer = KafkaProducer()

    def get_processed_bids(self):
        # se preiau toate ofertele procesate din topicul
        processed_bids_topic
        print("Aștept 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 folosește 2 *topic*-uri Kafka, astfel:

- **topic_oferte_procesate** → se așteaptă primirea ofertelor procesate de la **MessageProcessor**, iar pe măsură ce acestea sunt disponibile în topic, sunt preluate și adăugate într-o colecție internă.
- **topic_rezultat** → după decizia câștigătorului, mesajul efectiv trimis de acesta este publicat în **topic_rezultat**, pentru ca fiecare **Bidder** să îl preia de acolo și să afle cine a câștigat licitația.

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.

Creați un folder denumit, spre exemplu, **Auctioneer_Docker**, în care adăugați fișierul sursă al microserviciului **Auctioneer** (**Auctioneer.py**).

Alături de acel fișier sursă, creați un fișier **Dockerfile**, în care adăugați următorul conținut:

```

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"]

```

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

Execuția aplicației

Se recomandă execuția de la terminal, așa încât deschideți unul în folder-ul proiectului și creați un mediu virtual Python în care instalați modulul **kafka-python**:

```

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

```

Nu uitați să verificați că server-ul Kafka și containerul Docker cu ETCD sunt pornite!

Pregătirea topic-urilor

Înainte de fiecare pornire a aplicației, folosiți următorul script Python pentru a pregăti *topic*-urile Kafka pentru pornirea licitației:

• **PrepareAuction.py**

```

from kafka import KafkaAdminClient
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 = admin.list_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.")

```

Atenție: înainte de a-l executa, asigurați-vă că nu există niciun consumator Kafka pornit (niciun microserviciu în așteptare de mesaje, niciun script ce folosește API-ul Kafka pentru consumarea mesajelor din *topic*-urile folosite de aplicație).

Script-ul va șterge *topic*-urile existente pentru a forța golirea mesajelor existente de la execuții anterioare ale aplicației. Apoi, *topic*-urile vor fi recreate cu parametrii necesari.

```

/home/cosmin/venv/Licitatie/bin/python "/mnt/hdd/PREDARE/SD/laboratoare/laborator_10/0kazii_v2/PrepareAuction.py"
Se sterg topic-urile existente...
  Se sterge topic_oferte_procesate...
  Se sterge topic_oferte...
  Se sterge topic_notificare_procesor_mesaje...
  Se sterge topic_rezultat...
Se creeaza topic-urile necesare:
  topic_oferte
  topic_rezultat
  topic_oferte_procesate
  topic_notificare_procesor_mesaje
Gata! Microserviciile participante la licitatie pot fi pornite.

```

Figura 13 - Pregătire *topic*-uri Kafka

Dacă primiți o eroare cum că există deja un anumit *topic*: măriți timpul de așteptare pentru ștergere (parametrul `time.sleep(...)`). Verificați să nu fie pornit niciun consumator de mesaje din acele *topic*-uri conținute în tupla `used_topics`.

Dacă nu rulați script-ul înainte de aplicatie, microserviciile vor reciti mesajele anterioare, deoarece sunt persistente până la expirare. Puteți profita de această proprietate a Kafka pentru testarea independentă, izolată a microserviciilor: de exemplu, dacă porniți doar **MessageProcessor** și există mesaje din execuții anterioare, acesta va reciti ofertele existente în Kafka și le va procesa din nou, trimițând un alt set de mesaje procesate pentru **BiddingProcessor**.

Pornirea microserviciilor

Deoarece prin varianta de implementare cu Kafka s-a obținut un grad mare de decuplare, iar fiecare microserviciu așteaptă mesajele necesare lui pentru a funcționa, **microserviciile se pot porni în orice ordine se dorește**, cu mențiunea că **fiecare instanță a microserviciului Auctioneer așteaptă maxim 15 secunde de inactivitate din partea Bidder-ilor**.

Deci, se recomandă să porniți **Auctioneer** ca ultimul microserviciu.

Pornirea microserviciilor care nu sunt încapsulate în imagini Docker

Deschideți câte un terminal în folder-ul cu fișierele sursă ale proiectului, și porniți 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 pornește folosind **Docker Compose**, pentru a putea fi scalat cu ușurință: astfel, se pot porni oricâte instanțe **Auctioneer** dorește utilizatorul.

Creați un fișier numit **docker-compose.yml** în folder-ul **Auctioneer_Docker** creat anterior, și adăugați următorul conținut:

```

version: "3.4"
services:
  auctioneer:
    build:
      context: .
    network_mode: host

```

Deci, pentru a porni **2 instanțe** ale microserviciului **Auctioneer**, executați următoarea comandă în folder-ul **Auctioneer_Docker**, în care se regăsește fișierul **docker-**

compose.yml:

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

Un exemplu de execuție se regăsește în figura următoare:

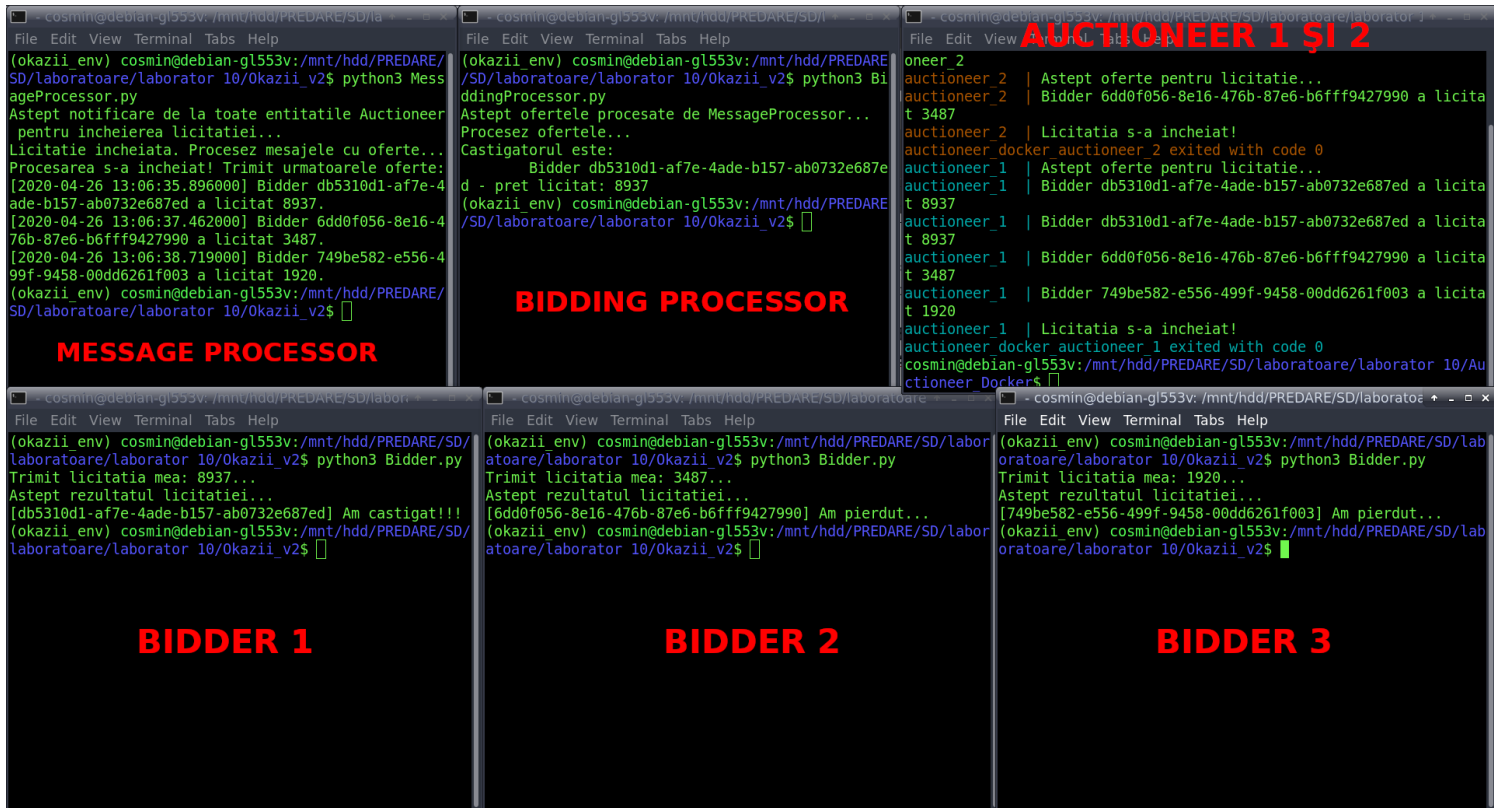


Figura 14 - Exemplu de execuție

Puteți observa împărțirea mesajelor pe partiții în cazul entităților **Auctioneer** și, de asemenea, existența unui mesaj duplicat care a fost filtrat de **MessageProcessor**.

Utilizarea Kafka pentru monitorizarea aplicațiilor în dev-ops

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

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

În scopul monitorizării aplicației **Okazii** din laborator, creați o aplicație separată Spring Boot (numită, de exemplu, **MonitoringApplication**), la care adăugați dependența **Spring-Kafka** (ca în exemplul 2 din laborator). Această aplicație reprezintă un microserviciu utilizat pentru instrumentație și 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(
        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 =
URLLEncoder.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("$BASE_DOCKER_API_COMMAND/containers/$auc-
tioneerContainerID/stats?stream=0")
            val auctioneerContainerStatsProcessInput =
BufferedReader(InputStreamReader(auctioneerContainerStatsProcess.in-
putStream))

            // 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?.get(1).let {
        println("[${auctioneerContainerID}] Utilizare memorie: $it")
    }
}
}
}

```

Componenta de monitorizare conține:

- un *listener* pentru Kafka, care urmărește 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 partițiile care vor fi urmărite, respectiv *offset*-ul de la care se începe citirea mesajelor (0 = de la început). Fiind aplicație de monitorizare, toate cele 4 partiții de la *topic*-ul **topic_oferte** sunt de interes. *Listener*-ul contorizează mesajele primite, în timp real.
- o funcție recurentă care afișează numărul de mesaje preluate de *listener*-ul Kafka: **showKafkaStats()**
- o funcție recurentă care monitorizează instanțele microserviciului **Auctioneer** sub formă de containere Docker (indiferent de numărul lor, în mod dinamic): **showAuctioneerContainersStats()**. Funcția 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** → afișează informații despre containerele în execuție. Se pot filtra rezultatele cu parametrul **filters**.
 - **/containers/<ID_CONTAINER>/stats** → afișează metrice despre un anumit container. Pentru a primi doar ultimele valori (și nu un flux continuu), se folosește 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

funcțiile recurente din Spring (adnotate cu `@Scheduler`).

Utilizați aplicația de monitorizare pentru a prelua metrice în timp real din aplicația Okazii. Porniți **MonitoringApplication** după ce a început licitația, și veți putea găsi metricele afișate în *log*-urile Spring:

```
[2020-04-26T23:14:55.157] Grad incarcare Auctioneer: 7 oferte primite
[2020-04-26T23:14:55.158] Grad incarcare MessageProcessor: 5 oferte procesate
[2020-04-26T23:14:56.158] Grad incarcare Auctioneer: 7 oferte primite
[2020-04-26T23:14:56.158] Grad incarcare MessageProcessor: 5 oferte procesate
[2020-04-26T23:14:57.159] Grad incarcare Auctioneer: 7 oferte primite
[2020-04-26T23:14:57.159] Grad incarcare MessageProcessor: 5 oferte procesate
[0e6372067795] Utilizare procesor: 181941570
[0e6372067795] Utilizare memorie: 11595776
[87d323d07ae1] Utilizare procesor: 262941315
[87d323d07ae1] Utilizare memorie: 11649024
[78f06d1293e5] Utilizare procesor: 242057484
[78f06d1293e5] Utilizare memorie: 11657216
[2020-04-26T23:15:02.366] Grad incarcare Auctioneer: 7 oferte primite
[2020-04-26T23:15:02.366] Grad incarcare MessageProcessor: 5 oferte procesate
[2020-04-26T23:15:03.366] Grad incarcare Auctioneer: 7 oferte primite
[2020-04-26T23:15:03.367] Grad incarcare MessageProcessor: 5 oferte procesate
[0e6372067795] Utilizare procesor: 272503533
[0e6372067795] Utilizare memorie: 11907072
[87d323d07ae1] Utilizare procesor: 351367441
[87d323d07ae1] Utilizare memorie: 11677696
[78f06d1293e5] Utilizare procesor: 330037046
[78f06d1293e5] Utilizare memorie: 11698176
[2020-04-26T23:15:09.439] Grad incarcare Auctioneer: 7 oferte primite
[2020-04-26T23:15:09.439] Grad incarcare MessageProcessor: 11 oferte procesate
[2020-04-26T23:15:10.440] Grad incarcare Auctioneer: 7 oferte primite
[2020-04-26T23:15:10.440] Grad incarcare MessageProcessor: 11 oferte procesate
```

Figura 15 - Monitorizarea aplicației Okazii

Valorile pentru utilizarea procesorului și a memoriei sunt de fapt valori brute raportate de sistemul de fișiere 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

Aplicații și teme

Teme de laborator

1. Încapsulați și microserviciul **Bidder** într-o imagine Docker. Creați Dockerfile pentru acesta și folosiți comanda **docker build** (exemplu de folosire în **laboratorul 8**) pentru construirea imaginii.
2. Utilizând **Docker Compose**, porniți un număr mare de astfel de microservicii (de exemplu, 100) la o execuție a aplicației. Scalați microserviciul **Auctioneer** la un număr de 4 instanțe și verificați că aplicația face față fluxului mare de mesaje.

Teme pentru acasă

1. Completați aplicația de monitorizare a resurselor astfel încât să afișeze un grafic în timp real pentru resursele monitorizate.

Puteți 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. Implementați licitația engleză (eng. *English auction*), pornind de la exemplul de aplicație din laborator (**Okazii v2**). Adăugați un nou procesor de flux care să implementeze logica necesară pentru regulile licitației engleze și integrați acest procesor în fluxul existent în așa fel încât utilizatorul să poată să aleagă ce tip de licitație dorește: tipul inițial de licitație, cu plic închis, sau tipul nou implementat, de tip *English auction*.

Găsiți regulile licitației engleze 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 și 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]: Documentație Spring Kafka (**codul din documentație 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 fișier 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/>