PROGRAMMAZIONE ORIENTATA AI MICROSERVIZI

Comunicazione tra microservizi

Tommaso Nanu
tommaso.nanu@alad.cloud

UNIVERSITÀ DI PARMA
DIPARTIMENTO DI SCIENZE MATEMATICHE, FISICHE E INFORMATICHE
Corso di Laurea in Informatica

29 novembre 2023

Presentazione di Jonathan Ambri
jonathan.ambri@alad.cloud

Comunicazione in un'architettura a microservizi

- Tra gli aspetti più complicati quando si passa da un’architettura *monolitica* a una a *microservizi* abbiamo il cambiamento del meccanismo di comunicazione.
  - In un’applicazione *monolitica* (in esecuzione su un singolo processo), tutta (o quasi) la logica di business risiede all’interno della stessa applicazione, quindi per comunicare tra componenti diversi possiamo limitarci ad invocare metodi del linguaggio (*intra-process communication*).
  - Un'applicazione basata su *microservizi* è un *sistema distribuito* in esecuzione su più servizi (container docker). Ogni istanza di servizio è in genere un processo. Pertanto, i servizi devono interagire utilizzando un protocollo di comunicazione tra processi (*inter-process communication*) come *HTTP*, *AMQP* (*RabbitMQ*), *Apache Kafka*, *gRPC* o un protocollo binario come *TCP*, a seconda della natura di ciascun servizio.

Tipi di comunicazione

- La prima distinzione è tra protocollo *sincrono* e asincrono:
  - *Protocollo* ***sincrono***: **HTTP.** Il client invia una richiesta e attende una risposta dal servizio. Questo è indipendente dall'esecuzione del codice client che potrebbe essere *sincrona* (il thread è bloccato) o *asincrona* (questo è il caso dell'utilizzo del pattern *async/await* in cui thread non è bloccato e alla ricezione della risposta può essere invocata una callback). Il client può proseguire con la propria esecuzione solo quando riceve la risposta HTTP dal servizio chiamato.
  - *Protocollo* ***asincrono***: per esempio, **Apache Kafka** o **RabbitMQ**. Vengono utilizzati messaggi asincroni. Il client che invia il messaggio verso il *message broker* non attende la risposta (il risultato).

    Il *message broker* è un intermediario che si occupa di coordinare la comunicazione tra più entità.

Tipi di comunicazione

- La seconda distinzione tra comunicazione con *un singolo ricevitore* e comunicazione *con più ricevitori*:
  - **Single receiver**: Ogni richiesta deve essere elaborata esattamente da un solo destinatario (servizio). Questo tipo di comunicazione può essere sincrona o asincrona. Un esempio è l'utilizzo del protocollo sincrono HTTP quando si richiama una Web API.
  - **Multiple receivers**: Ogni richiesta può essere elaborata da zero a più ricevitori. Questo tipo di comunicazione deve essere asincrona. Un esempio è il meccanismo di *publish/subscribe* utilizzato in pattern come l'*Event-driven architecture* (*EDA*), che utilizzano il message broker per la propagazione di un messaggio a più ricevitori.

Tipi di comunicazione

- Un'applicazione basata su microservizi utilizzerà spesso una combinazione di questi stili di comunicazione.
- Idealmente, bisogna cercare di ridurre al minimo la comunicazione *sincrona* tra i microservizi, in modo da ridurre le *dipendenze* tra di essi. L'obiettivo di ciascun microservizio è quello di essere autonomo e disponibile ad un client, anche se altri servizi che fanno parte dell'applicazione non sono disponibili per una qualche ragione.
- Quanto più si aggiungono dipendenze sincrone tra microservizi, come le richieste di query tramite protocollo HTTP, tanto peggiore diventa il tempo di risposta complessivo.
- Il tutto peggiora ancora di più quando si creano delle catene di chiamate sincrone HTTP.

Sovranità dei dati e replica

- Se un servizio necessita di dati che risiedono su altri servizi, bisogna evitare di effettuare richieste sincrone per tali dati. Conviene replicare (tramite propagazione) i dati (solo le informazioni necessarie) nel database del servizio iniziale, tramite l'utilizzo di *messaggi asincroni*.

  L'esigenza di mantenere allineati database di differenti servizi nasce dal fatto che ci sono tabelle che devono essere «condivise» tra servizi diversi: nel modello a microservizi però, ogni servizio possiede il proprio database con le proprie tabelle per cui per mantenere una tabella condivisa tra più servizi conviene replicarla sui database di tutti i servizi di cui ne hanno bisogno (in certi casi con un set di colonne più limitato a determinate esigenze).

  I vari database che condividono la tabella devono però essere allineati ai valori contenuti nella tabella "master" definita nel database del servizio di riferimento, che si occupa di mantenere i dati più aggiornati e completi per la relativa tabella.
- È possibile utilizzare *Apache Kafka* per mantenere «sincronizzati» i database di servizi differenti, attraverso un pattern chiamato *Transactional Outbox*.

Comunicazione asincrona con Apache Kafka

(Publish - Subsribe messaging system)

29 novembre 2023

Kafka

“Apache Kafka is a distributed event store and stream-processing open-source platform”

- Kafka è una piattaforma per il *data-streaming* (flussi di dati generati continuamente da diverse fonti) *distribuita* che permette di *pubblicare*, *sottoscrivere*, *archiviare* ed *elaborare* flussi di record in tempo reale.
- È progettato per gestire flussi di dati provenienti da più sorgenti (*Producers*) distribuendoli a più consumatori (*Consumers*).
- Kafka è *open-source* ed è sviluppato dalla *Apache Software Foundation* ed è scritto in *Java* e *Scala*.
- Kafka utilizza un *protocollo binario* basato su *TCP*.

Kafka: Message broker

- Tra le capacità fondamentali di Kafka, abbiamo quella di supportare il pattern **publish - subscribe** per la trasmissione di stream di record (flussi di messaggi), ovvero è in grado di agire da **message broker** per la comunicazione **asincrona** (utilizzo per la messaggistica asincrona):
  - I broker di messaggi possono mantenere dei buffer di messaggi non elaborati.

Kafka: Componenti

Le Componenti principali di Kafka sono:

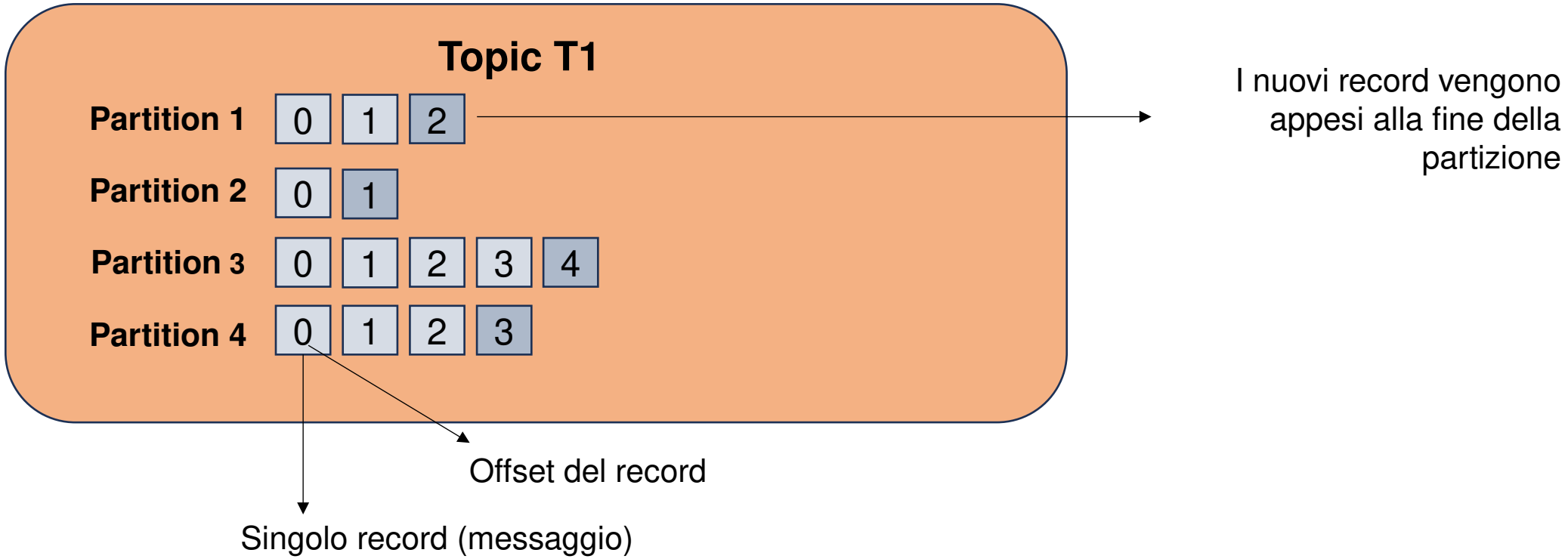
- **Topic**: è una sorta di categoria, identificata da un nome, utilizzata per raggruppare i messaggi; un Consumer si *sottoscrive* ad un *topic* per ricevere i messaggi di quella particolare categoria, mentre un Producer invia un messaggio verso un determinato topic;
- **Partition** (partizione): ciascuna delle sottosezioni in cui è diviso un topic;
- **Record**: è il *messaggio* vero e proprio, costituito da una *chiave*, un *valore* e un *timestamp*;
- **Offset**: indice che identifica univocamente un record all'interno di una partizione;
- **Producer** (produttore): entità che invia i messaggi a Kafka;
- **Consumer** (consumatore): entità che riceve i messaggi da Kafka;
- **Group** (gruppo): è un'etichetta utilizzata per distinguere *insiemi di Consumer* sottoscritti a un topic;
- **Broker**: è un processo (nodo) che si occupa di gestire la ricezione e il salvataggio dei messaggi e i relativi *offset*;
- **Cluster**: insieme di più Broker utile per replicare e distribuire le partizioni; nel seguito, un Cluster sarà chiamato semplicemente Kafka.

Kafka: Producer e Consumer (Clients)

- Producer e consumer agiscono come dei ***client*** di Kafka.
- La comunicazione tra Kafka e i suoi client avviene mediante un *protocollo binario* basato su *TCP.*
- Un client può essere sia producer che consumer.
- Più producer possono *pubblicare* messaggi sullo stesso *topic*; e allo stesso modo, più consumer possono *sottoscriversi* allo stesso *topic*.
- Kafka ignora il contenuto dei messaggi, che possono essere nel formato preferito, per esempio in *JSON.*

Kafka: Topic e Partizioni

- Un **topic** può essere suddiviso in più *partizioni* (almeno una).
- Una **partizione** è una sequenza ordinata e immutabile di record (messaggi/eventi).
- Ogni nuovo record relativo ad un topic, viene appeso su una sola partizione specifica.
- Ogni record all'interno di una partizione ha uno specifico **offset**, che lo identifica in modo univoco nella partizione.
- Vediamo un topic con più partizioni:



Kafka: Consumer e Gruppo

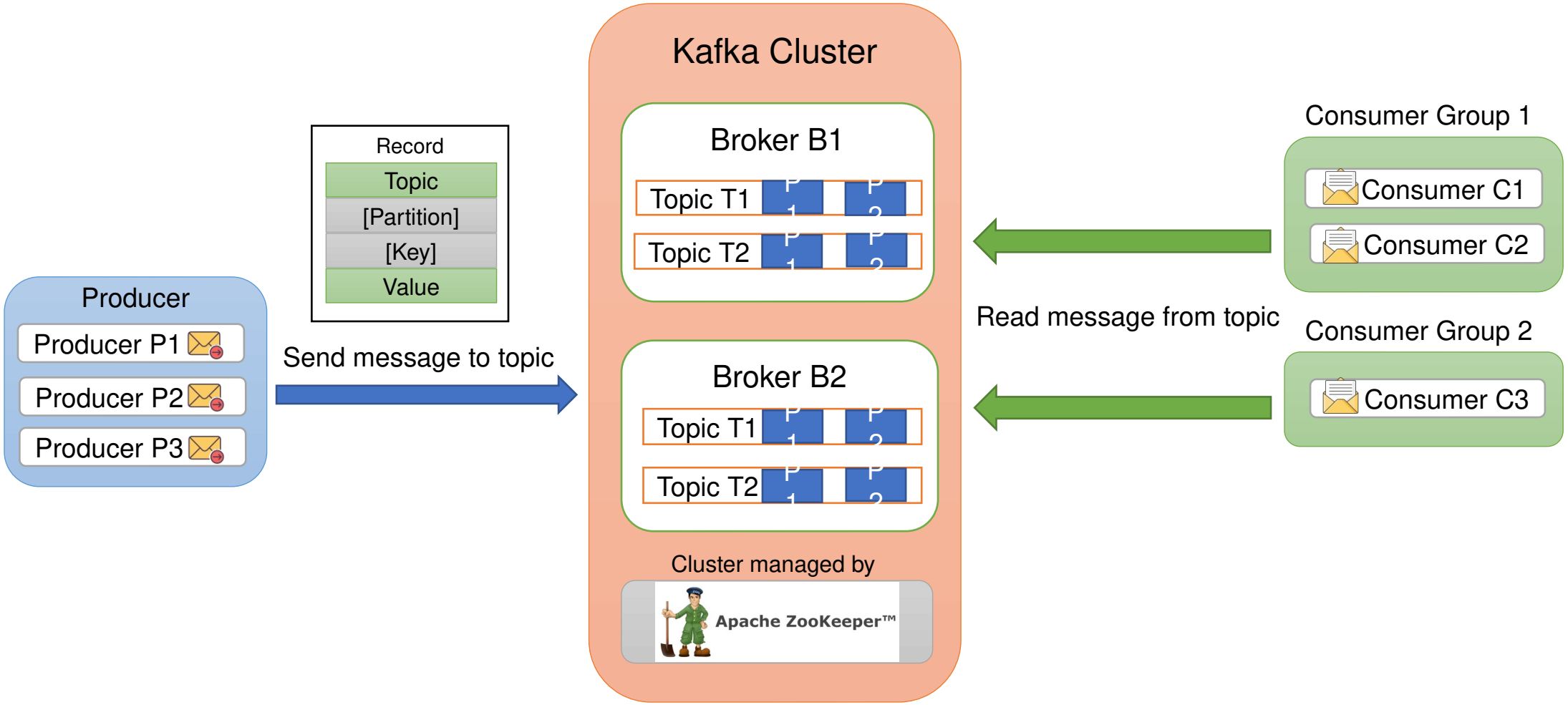
- Quando un *consumer* si sottoscrive a un *topic*, deve specificare anche il relativo **gruppo,** che viene utilizzato da Kafka per la distribuzione dei record ai vari consumatori abbonati al topic.
- I messaggi pubblicati su un topic vengono consegnati a un solo consumer per ciascun gruppo:

  ovvero il messaggio viene consegnato a tutti i gruppi ma a un solo consumer per ciascun gruppo.
- Kafka assegna (dinamicamente) zero, una o più partizioni del topic a ciascun consumer presente in un gruppo, e consegna tutti gli eventi di quelle partizioni a quel determinato consumer.
- Se in un gruppo, il numero dei consumer è maggiore del numero delle partizioni del topic, allora alcuni consumer del gruppo non riceveranno nessun messaggio dal topic.

Kafka: Ordine dei messaggi

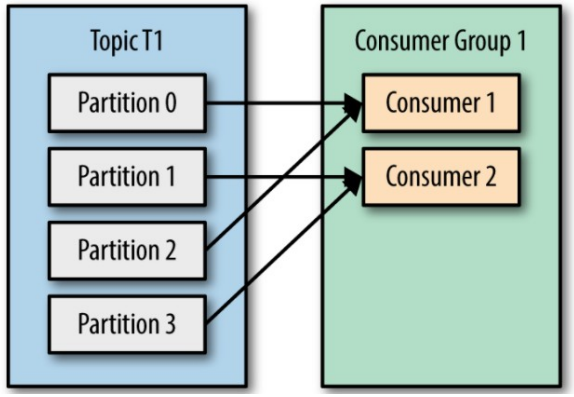
- I record pubblicati da un producer su un topic verranno appesi alle rispettive partizioni nell'ordine in cui sono stati pubblicati.
- Un consumer riceverà i messaggi da una partizione di un topic nell'ordine in cui sono memorizzati nella partizione.
- Se abbiamo un topic, con una sola partizione, un solo producer e un solo consumer, i vari messaggi verranno ricevuti dal consumer nell'ordine in cui sono stati pubblicati dal producer, tuttavia, questo non è garantito se il topic è composto da più partizioni.
- Tuttavia, l'uso di una sola partizione per topic non consente di distribuire i messaggi ai vari consumer (se sono più di uno) presenti in un gruppo.

Kafka: Architettura

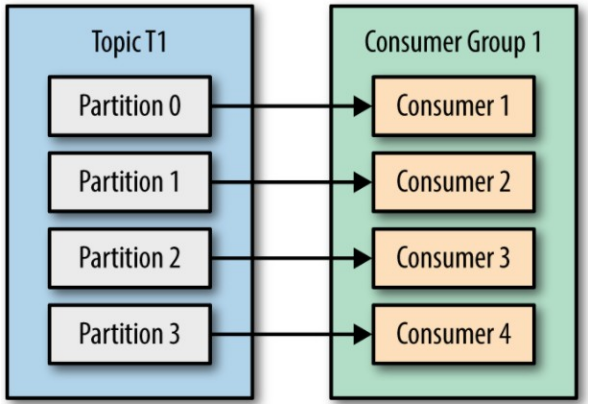


Kafka: Partizioni e Consumatori

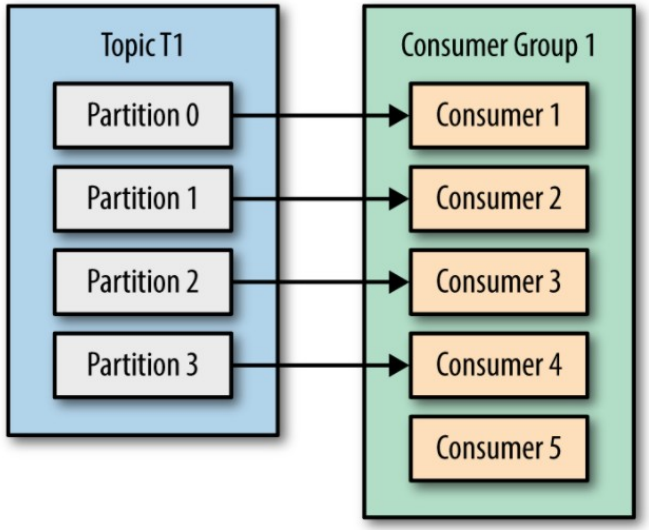
In un **gruppo** di consumatori, ogni **partizione** verrà elaborata da un solo consumatore . Questi sono gli scenari possibili:



Il numero di consumatori all'interno del gruppo è **inferiore** al numero di partizioni del topic, quindi è possibile assegnare più partizioni a uno dei consumatori nel gruppo.

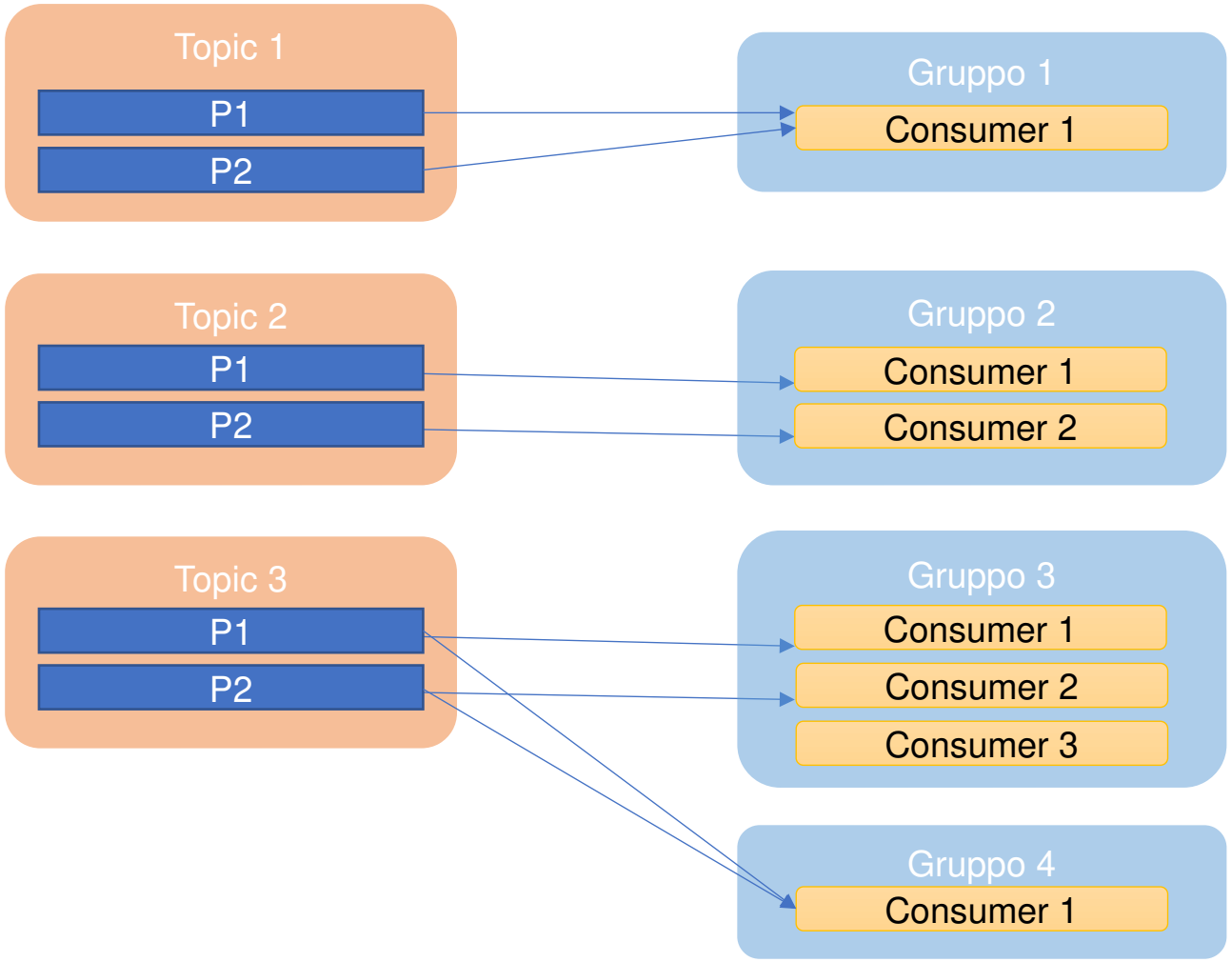


Il numero di consumatori all'interno del gruppo è **uguale** al numero di partizioni del topic, quindi ogni partizione può essere associata ad un solo consumatore.



Il numero di consumatori all'interno del gruppo è **maggiore** del numero di partizioni del topic, quindi alcuni consumatori non riceveranno alcun messaggio perché non hanno una partiziona associata.

Kafka: Partizioni e Consumatori



In un gruppo di consumatori, ogni partizione di un topic verrà elaborata da un solo consumatore (a parità di gruppo) .

Kafka: Consumer auto.offset.reset

- La proprietà **auto.offset.reset** (property **AutoOffsetReset** del **ConsumerConfig**) del consumer consente di regolare gli aspetti temporali della consegna dei messaggi:
  - Il valore **earliest** (**smallest**) specifica che il consumer deve ricevere tutti i messaggi pubblicati sul topic, compresi quelli pubblicati in passato, prima della sua sottoscrizione.

    Viene impostato l'offset di riferimento del consumer sul valore più piccolo.
  - Il valore **latest** (**largest**) specifica che il consumer deve ricevere solo i messaggi pubblicati sul topic dal momento in cui si è sottoscritto, escludendo tutti quelli pubblicati in precedenza.

    Viene impostato l'offset di riferimento del consumer sul valore più grande.

Kafka: API per la comunicazione asincrona

Delle API fornite da Kafka, noi utilizzeremo solo quelle utili per gestire la comunicazione asincrona tra servizi:

- **Admin API**: utilizzata per gestire *topic* e *broker;*
- **Producer API**: utilizzata dai producer per pubblicare un record su uno o più topic;
- **Consumer API**: utilizzata dai consumer per sottoscriversi ad uno o più topic e per ricevere i relativi messaggi.

- Per **.NET**, è disponibile il pacchetto NuGet **Confluent.Kafka**, che rappresenta il porting in *C#* per le API Admin, Producer e Consumer.

Installazione e configurazione di Kafka

- Nel cluster deve essere installato sia **Kafka** che **ZooKeeper**, che viene utilizzato per il coordinamento dei broker Kafka.

  https://kafka.apache.org/quickstart

- Noi utilizzeremo Kafka e ZooKeeper tramite *container Docker*: le relative immagini da scaricare sono:

  bitnami/kafka:3.1.0

  bitnami/zookeeper:3.9.0

docker-compose.yml

```
version: '3.4'
services:
  zookeeper:
    container_name: zookeeper
    image: 'bitnami/zookeeper:3.9.0'
    ports:
      - '2181:2181'
    environment:
      - ALLOW_ANONYMOUS_LOGIN=yes

  kafka:
    container_name: kafka
    image: 'bitnami/kafka:3.1.0'
    ports:
      - '9092:9092'
    environment:
      - KAFKA_BROKER_ID=1
      - KAFKA_CFG_LISTENERS=PLAINTEXT://:9092
      - KAFKA_CFG_ADVERTISED_LISTENERS=PLAINTEXT://kafka:9092
      - KAFKA_CFG_ZOOKEEPER_CONNECT=zookeeper:2181
      - ALLOW_PLAINTEXT_LISTENER=yes
      - KAFKA_AUTO_CREATE_TOPICS_ENABLE=true
    depends_on:
      - zookeeper
```

Installazione e configurazione di Kafka

- Mostrare i container in esecuzione:
  > ***docker ps***
- Mostrare tutti i container:
  > ***docker ps --all***
- Apre una nuova shell in un container esistente (per esempio nel container **kafka**)
  > ***docker exec -it kafka bash***

  > *cd /opt/bitnami/kafka/bin*
  > *cd /opt/bitnami/kafka/config*

Esempio Producer e Consumer

[Vedere i progetti Utility.Kafka e UniprExample]

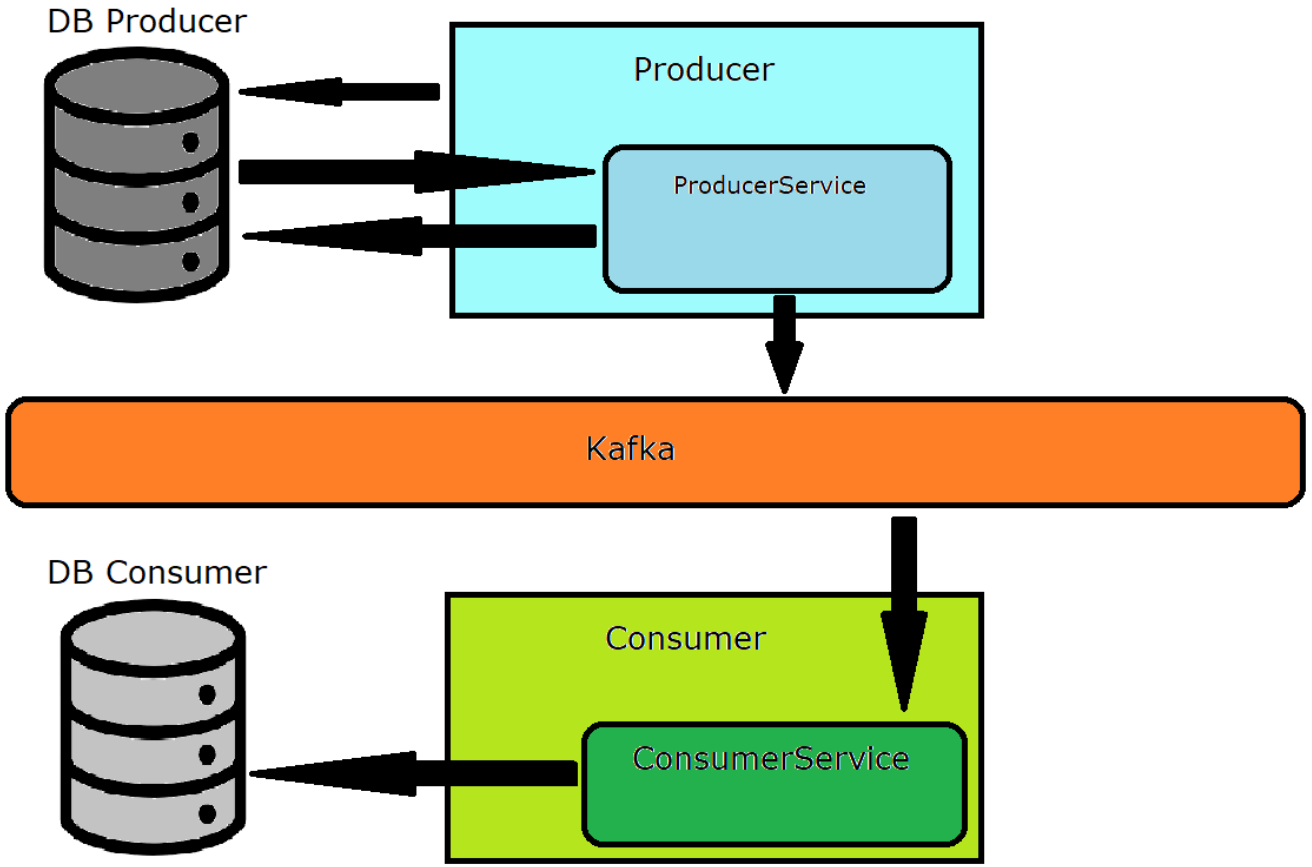
Transactional Outbox Pattern

- Il *Transactional Outbox Pattern* è implementato dal servizio *Producer*:

- Per ogni operazione di Insert/Update/Delete sulle tabelle *condivise*, deve essere eseguita una Insert sulla tabella *TransactionalOutbox*, inserendo nel campo *Messaggio* la serializzazione di un *OperationMessage<TDto>* (che contiene il Dto del record modificato e l'operazione eseguita) e nel campo Tabella il nome della tabella condivisa su cui è stata eseguita l'operazione.

- Ogni minuto (per esempio) viene poi eseguito il metodo *OperationsAsync* del *ProducerService* (definito sempre nel servizio *Producer*) che si occupa di verificare se ci sono record nella tabella *TransactionalOutbox*: per ogni record presente invia un messaggio *Kafka*, contenente il valore del campo *Messaggio*, verso il *topic* associato al campo *Tabella*. Una volta inviato il messaggio, il relativo record *TransactionalOutbox* viene eliminato.

Transactional Outbox Pattern



Comunicazione sincrona con HttpClient

29 novembre 2023

HttpClient

- **.NET** fornisce la classe ***HttpClient*** (namespace *System.Net.Http*) per poter comunicare in modo *sincrono* tra servizi, utilizzando il protocollo *HTTP*.

- Possiamo registrare un *HttpClient* tramite il metodo di estensione ***AddHttpClient*** sull'interfaccia *IServiceCollection*: questo metodo registra e utilizza un'interfaccia ***IHttpClientFactory*** per configurare e creare istanze di *HttpClient.*

Typed clients

- I *typed clients* permettono di configurare e interagire in un'unica posizione con un *HttpClient* e di renderlo disponibile in tutto il servizio tramite DI.
- Per definire un *typed client* creiamo una classe *NomeClient* che tra i parametri del costruttore ha un *HttpClient* (utilizzato per valorizzare un campo privato *HttpClient*):

```
public class NomeClient : INomeClient {
private readonly HttpClient _httpClient;
public TransactionRepBaseClient(HttpClient httpClient) {
 _httpClient = httpClient;
}
…
}
```

HttpClient

- Per registrare un *typed client INomeClient*, nel Program.cs possiamo aggiungere:

```
IConfigurationSection confSection =
builder.Configuration.GetSection(NomeClientOptions.SectionName);
NomeClientOptions options = confSection.Get<NomeClientOptions>() ?? new();
builder.Services.AddHttpClient<INomeClient, NomeClient>(o => {
    o.BaseAddress = new Uri(options.BaseAddress);
});
```

  - Registra un'istanza di *HttpClient* specificando la property ***BaseAddress***.
  - Registra un'istanza di *INomeClient* come transient, passando l'istanza di *HttpClient* al relativo costruttore.

Repository NuGet su GitHub

Lettura/Scrittura di pacchetti NuGet da/su repository GitHub

29 novembre 2023

Generazione PAT (Personal Access Token) classic

https://docs.github.com/en/packages/working-with-a-github-packages-registry/working-with-the-nuget-registry

- Creare un proprio Account GitHub;
- Visitare la pagina https://github.com/settings/tokens/new ;
- Assegnare il permesso ***read:packages*** (*Download packages from GitHub Package Registry*) oppure ***write:packages*** (*Upload packages to GitHub Package Registry*);
- Eventualmente, estendere la durata del token.

Configurazione Solution: NuGet.Config

- Creare un file *NuGet.Config* nella cartella contenente il file con estensione *.sln*, con il seguente contenuto:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <packageSources>
        <add key="nuget.org" value="https://api.nuget.org/v3/index.json" protocolVersion="3" />
        <add key="nomeSource" value="https://nuget.pkg.github.com/NomeAccount/index.json" protocolVersion="3" />
    </packageSources>
    <packageSourceMapping>
        <packageSource key="nuget.org">
            <package pattern="*" />
        </packageSource>
        <packageSource key="nomeSource">
            <package pattern="Utility.*" />
        </packageSource>
    </packageSourceMapping>
    <packageSourceCredentials>
        <nomeSource>
            <add key="Username" value="%NOMESOURCE_NUGET_USERNAME%" />
            <add key="ClearTextPassword" value="%NOMESOURCE_NUGET_PASSWORD%" />
        </nomeSource>
    </packageSourceCredentials>
</configuration>
```

Configurazione Solution: NuGet.Config

- La sezione ***<packageSources>*** aggiunge il repository NuGet alla lista di repository accessibili dalla solution. L'accesso al repository pubblico nuget.org non viene revocato.

- La sezione ***<packageSource key="…">*** indica a NuGet di utilizzare il repository indicato per fare pull dei pacchetti il cui nome rispetta le pattern indicate.

- La sezione ***<packageSourceCredentials>*** indica *username* e *password* di accesso, in questo caso da *variabili d'ambiente*.

.csproj per pacchetto NuGet

- Per generare un pacchetto NuGet aggiungere le seguenti properties al file di progetto ***.csproj***:

```
<Project Sdk="Microsoft.NET.Sdk">
        <PropertyGroup>
        <AssemblyVersion>1.0.0.0</AssemblyVersion>
        <PackageVersion>1.0.0</PackageVersion>
        <PackageReadmeFile>README.md</PackageReadmeFile>
        <TargetFramework>net7.0</TargetFramework>
        <ImplicitUsings>enable</ImplicitUsings>
        <Nullable>enable</Nullable>
        </PropertyGroup>
        <ItemGroup>
        <None Include="README.md" Pack="true" PackagePath="\" />
        </ItemGroup>
</Project>
```

- Aggiungere un file README.md al progetto (contenente una descrizione a piacere del pacchetto)

.csproj per pacchetto NuGet

- ***<AssemblyVersion>*** deve aumentare solo quando ci sono *breaking changes*, non deve essere modificato per le release di nuove minor versions. Per una libreria che non ha mai breaking changes, l'AssemblyVersion deve restare invariato per tutta la vita della libreria.

- ***<PackageVersion>*** deve rispettare le regole di **SemVer 2.0** (https://semver.org/). In sintesi, tre numeri nel formato ***MAJOR.MINOR.PATCH***: ***MAJOR*** aumenta quando ci sono *breaking changes*; ***MINOR*** aumenta quando vengono aggiunte funzionalità; ***PATCH*** aumenta quando vengono risolti bug e/o anomalie.

- Per impedire che un progetto sia trasformato in pacchetto NuGet, aggiungere ***<IsPackable>false</IsPackable>*** (il default è *<IsPackable>true</IsPackable>*).
- Bisogna aggiungere la seguente properties al *docker-compose* per ignorarlo in fase di creazione dei pacchetti:

  <Target Name="Pack"></Target>

Creazione e pubblicazione dei pacchetti NuGet

- > *dotnet **build** --configuration "Release" --no-restore "NomeSolution.sln"*

  Compilazione della soluzione
- > *dotnet **pack** --configuration Release "NomeSolution.sln"*

  Creazione dei pacchetti NuGet
- > *dotnet **nuget push** "src/*/bin/Release/*.nupkg" --source "nomeSource" --skip-duplicate*

  Pubblicazione dei pacchetti NuGet

- Per aggiungere un pacchetto NuGet presente su GitHub ad un progetto:

> *dotnet **add** NomeProgetto.csproj **package** NomePacchetto --version 1.0.0 -s https://nuget.pkg.github.com/NomeAccount/index.json*