



Zajęcia 10

Asynchroniczna komunikacja pomiędzy serwisami

autor: Michał Starosta

allegro

Agenda

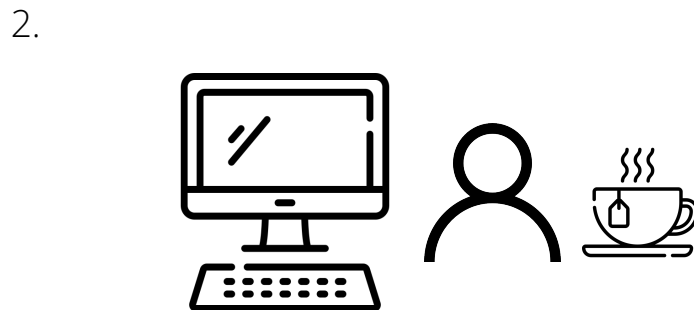
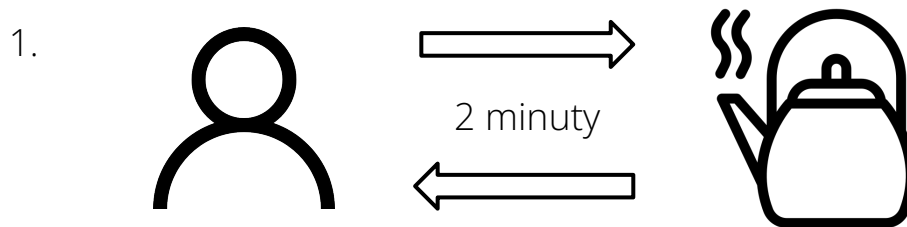
Agenda

- Komunikacja asynchroniczna w codziennym życiu
- Różnica między modelem synchronicznym, a asynchronicznym
 - plusy, minusy, typowe patterny
 - czym się różni kolejka od modelu pub-sub?
- Typowe problemy w komunikacji asynchronicznym
 - ostateczna spójność
 - exactly once
 - deduplikacja i kolejność zdarzeń
 - transakcyjność
 - poison message
- Technologie
 - RabbitMq
 - Kafka
 - Hermes

Komunikacja asynchroniczna w codziennym życiu

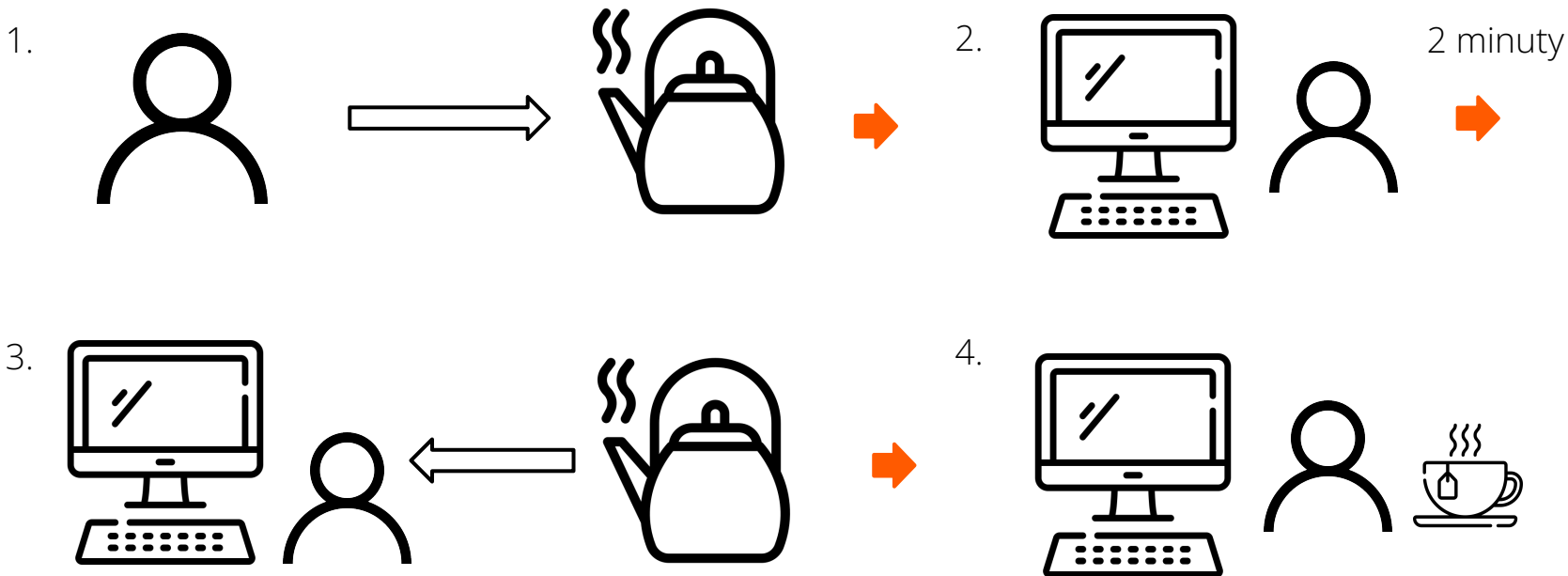
Zaparzanie herbaty w modelu synchronicznym

1. W oczekiwaniu na herbatę patrzymy się na czajnik.
2. Zanosimy herbatę do komputera.



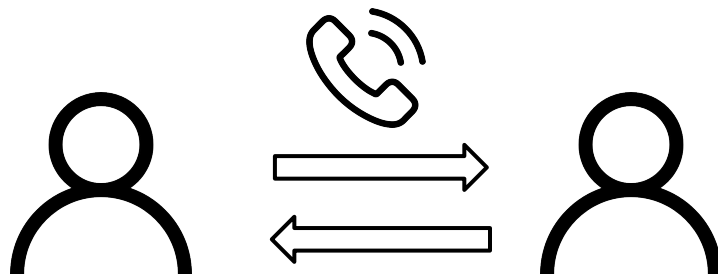
Zaparzanie herbaty w modelu asynchronicznym

1. Włączamy wodę na herbatę.
2. W oczekiwaniu na herbatę idziemy posiedzieć przy komputerze.
3. Czajnik informuje nas, że woda się zagotowała.
4. Zanosimy herbatę do komputera.



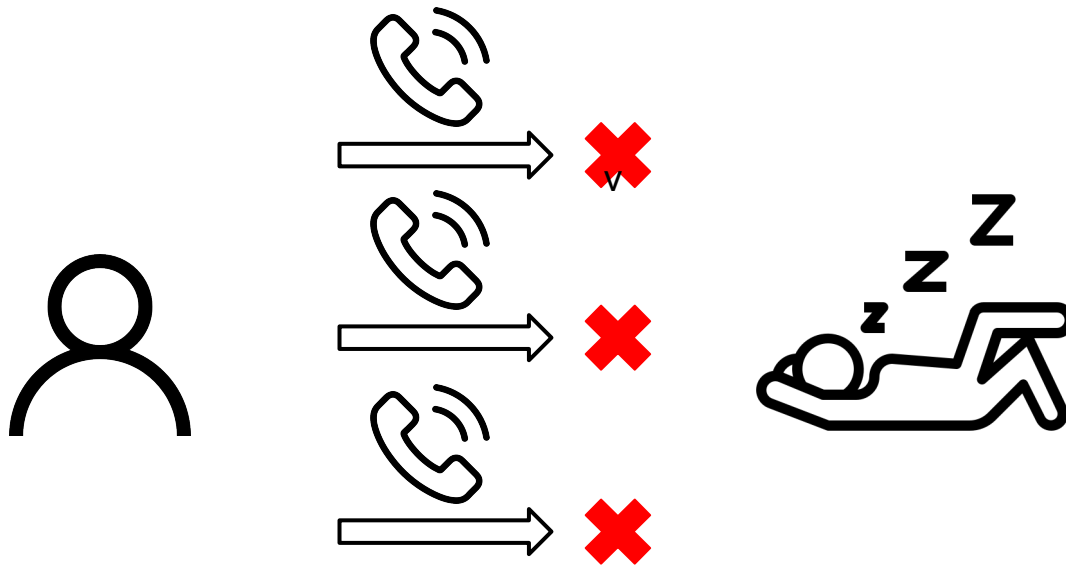
Rozmowa w modelu synchronicznym - przez telefon

Obie strony otrzymują informacje natychmiast



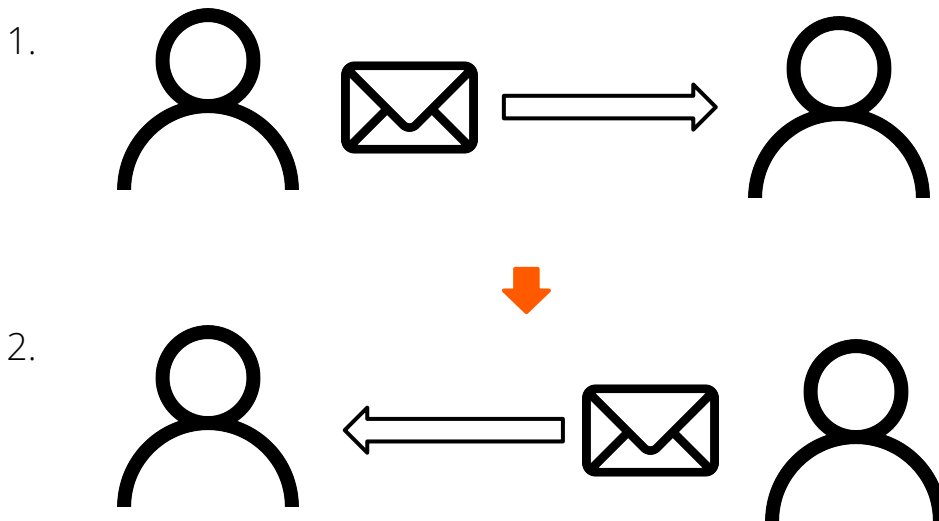
Rozmowa w modelu synchronicznym - przez telefon

Można się nie dodzwonić, nawet stosując powtórzenia



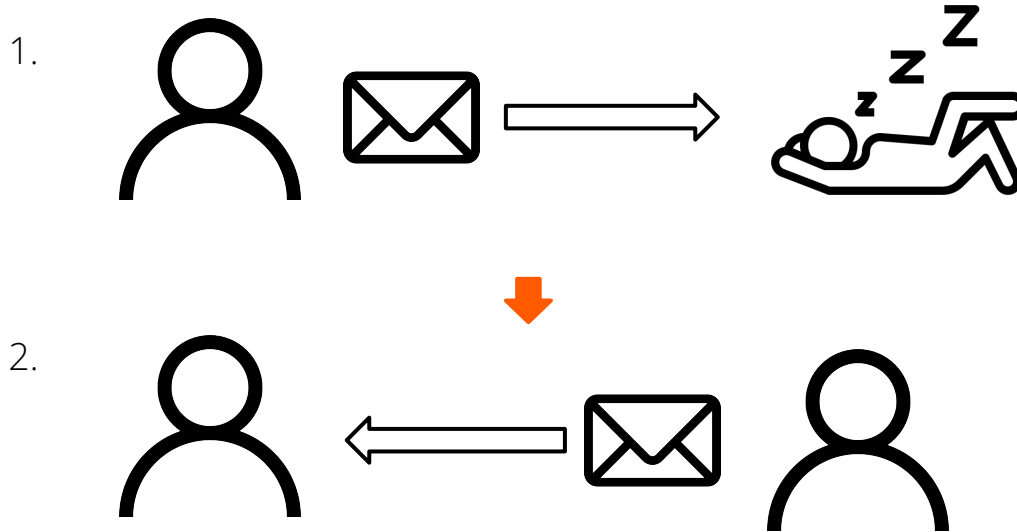
Rozmowa w modelu asynchronicznym - sms

Osoby wysyłają do siebie wiadomości niezależnie.



Rozmowa w modelu asynchronicznym - sms

Osoby wysyłają do siebie wiadomości niezależnie, mogą odczytać je później.

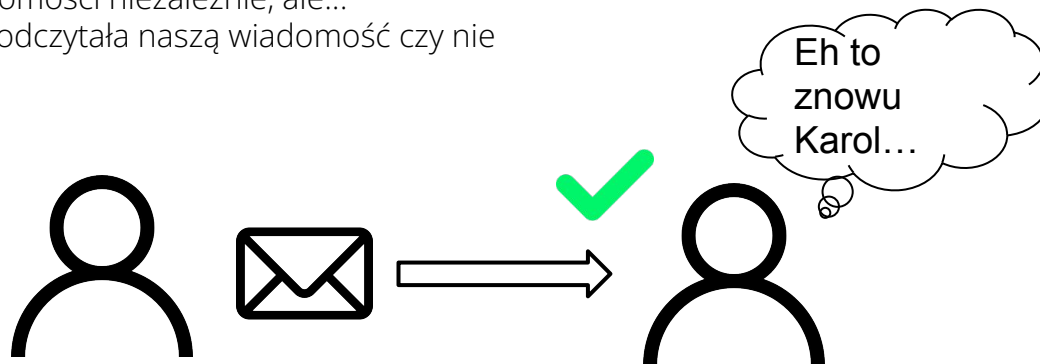


Rozmowa w modelu asynchronicznym - sms

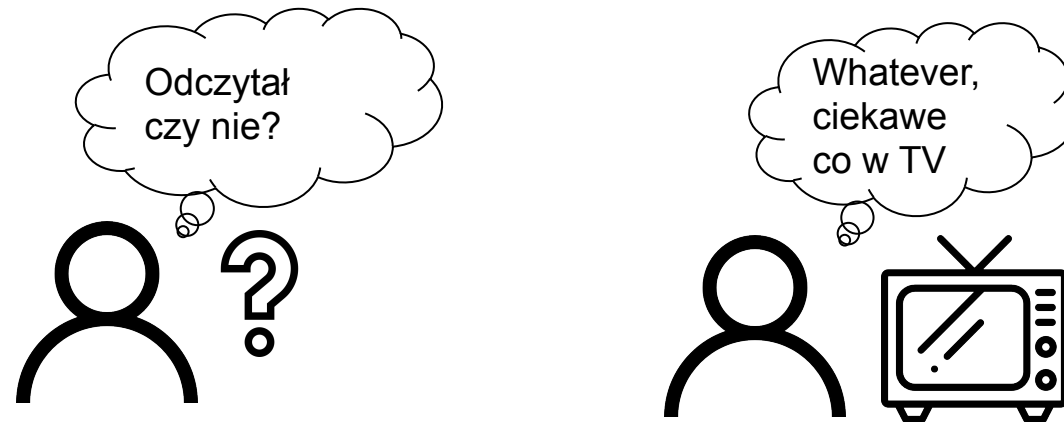
Osoby wysyłają do siebie wiadomości niezależnie, ale...

Nie wiadomo czy dana osoba odczytała naszą wiadomość czy nie

1.

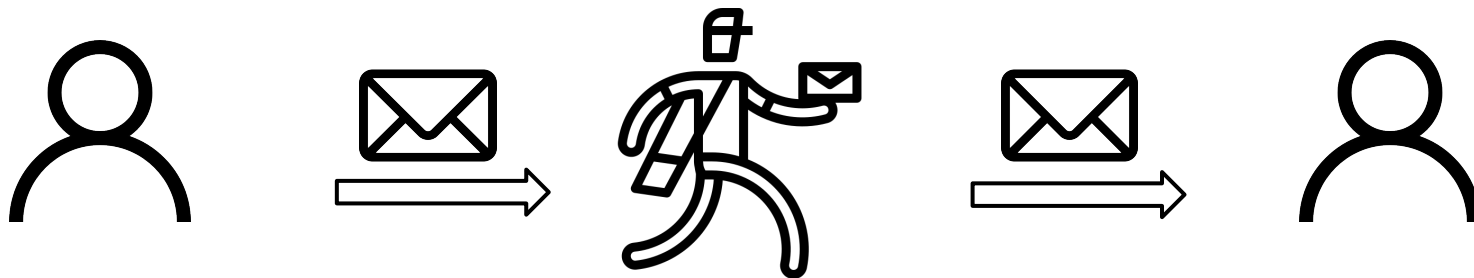


2.



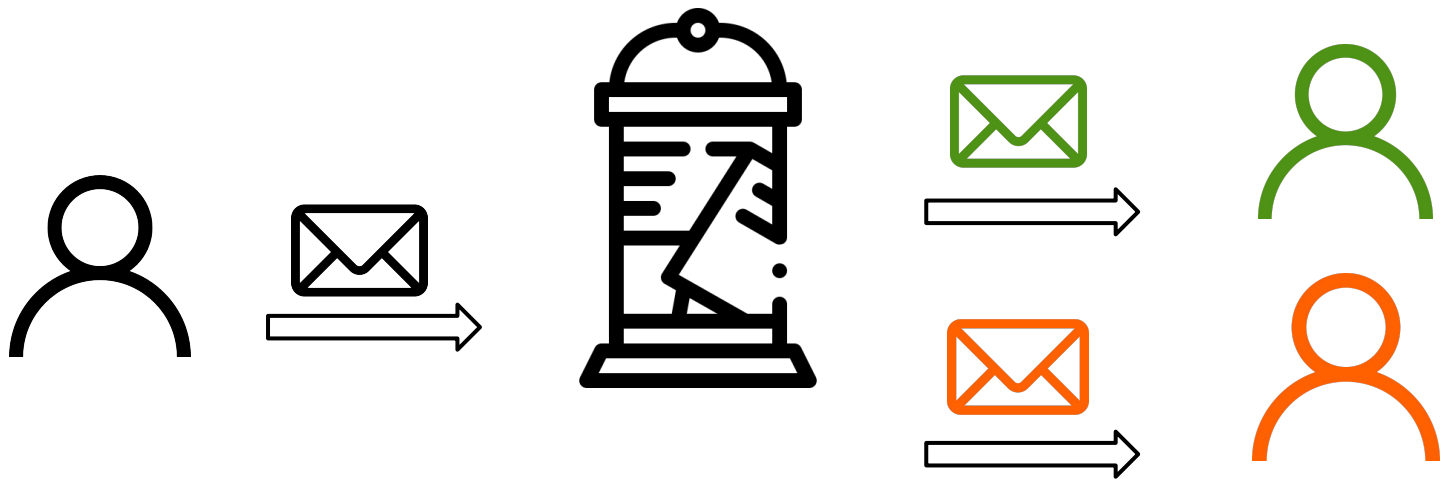
Komunikacja asynchroniczna, poczta

Osoby wysyłają do siebie wiadomości, ale za pośrednictwem, operatora lub listonosza.



Komunikacja asynchroniczna, ogłoszenia

Nadawca zostawia wiadomość na słupie ogłoszeniowym, odbiorcy sprawdzają słup w celu odczytania wiadomości.



Różnica między modelem
synchronicznym, a asynchronicznym

Komunikacja synchroniczna

Przykłady:

- Wywołanie endpointu
- REST
- HTTP
- RPC
- GRPC



Komunikacja asynchroniczna

Przykłady:

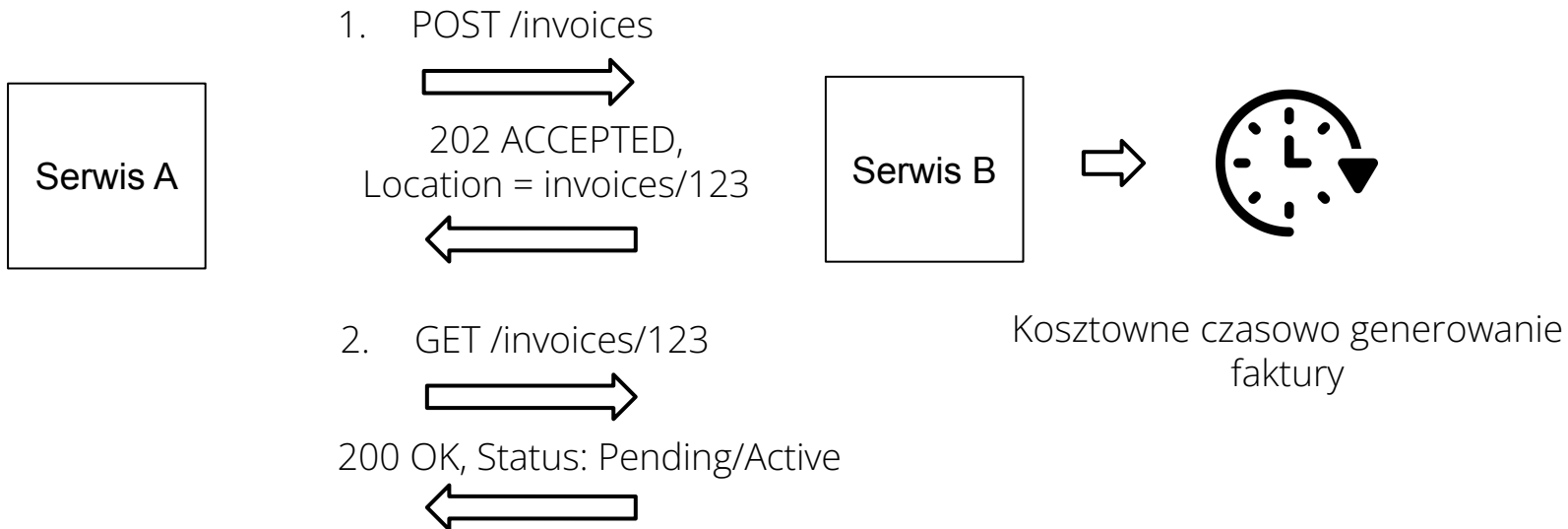
- Kolejka (Queue) - wiadomości
- Model pub-sub (publisher-subscriber) - zdarzenia
- Wywołanie endpointu (202 Accepted)



Pattern: Endpoint zwracający 202 Accepted

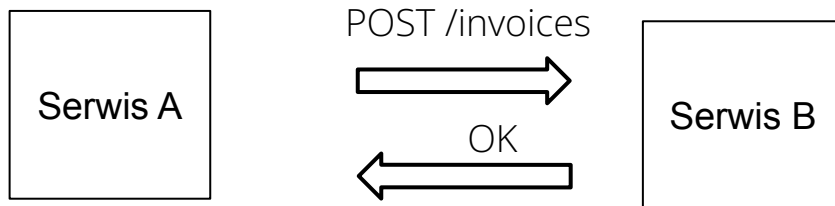
202- Status odpowiedzi informujący że zapytanie zostało zaakceptowane ale jeszcze nie przeprocesowane. Status wykorzystywany jest dla operacji z natury asynchronicznych. Zwraca nagłówek **location** z informacją gdzie pytać o status procesowanego zasobu.

Da się zasymulować asynchroniczność za pomocą synchronicznego protokołu http.

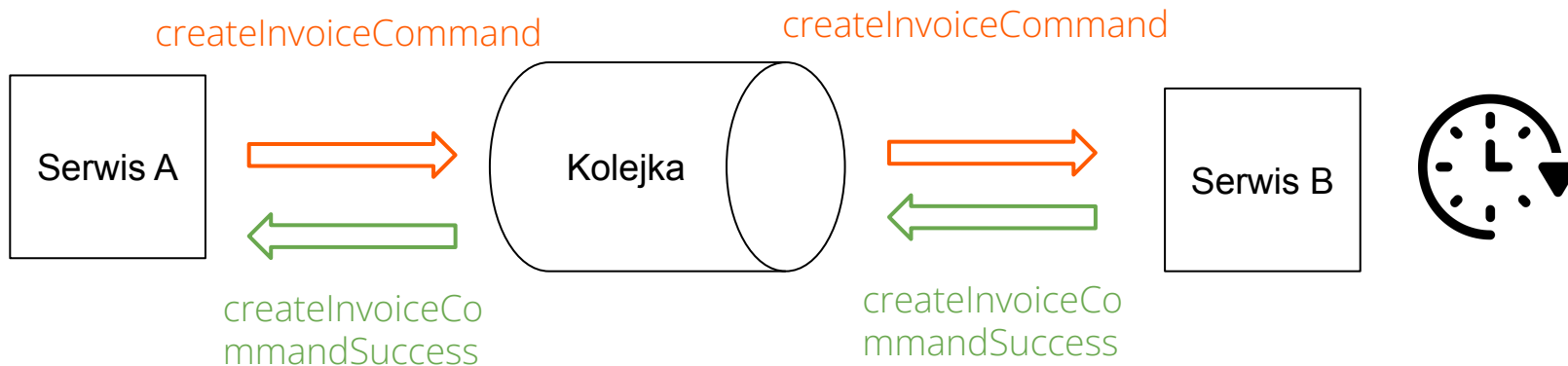


Pattern: Wzorzec komendy z wykorzystaniem kolejki

W celu zmniejszenia powiązania między serwisami zastępujemy zapytania synchroniczne wiadomościami na kolejce - wzorzec komendy. Komendy typowo służą do przekazywania wiadomości pojedynczemu odbiorcy.



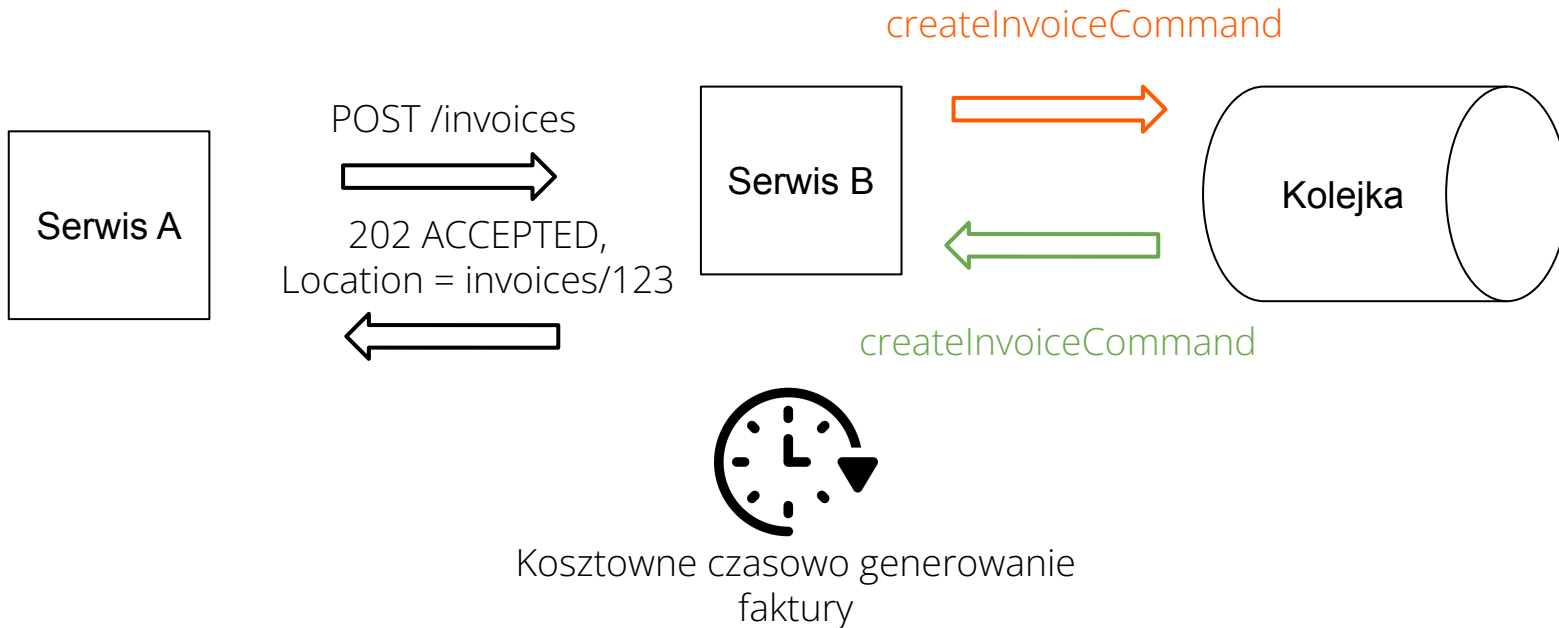
Komunikacja synchroniczna



Komunikacja asynchroniczna

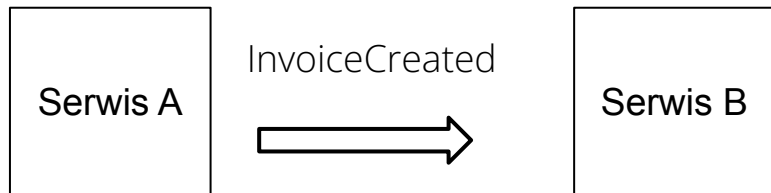
Pattern: Wzorzec komendy z wykorzystaniem kolejki

Kosztowne i długie operacje które nie zdążyłyby się wykonać w czasie zapytania synchronicznego możemy wykonać asynchronicznie z pomocą kolejki



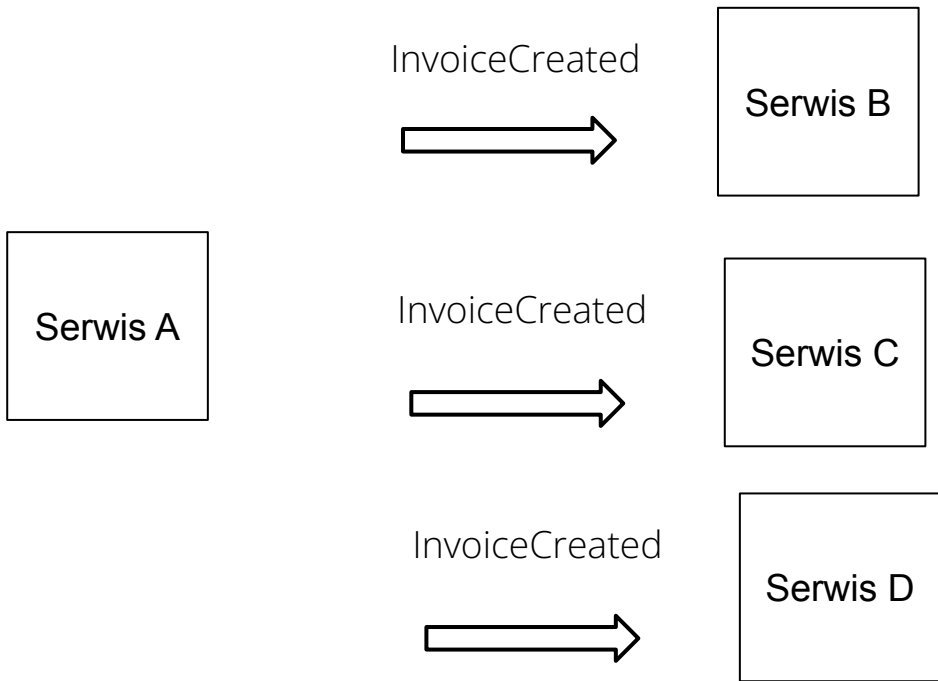
Pattern: Zdarzenia z wykorzystaniem pub-sub

Serwis synchronicznie powiadamia inny serwis o zmianie w wyniku jakiejś operacji



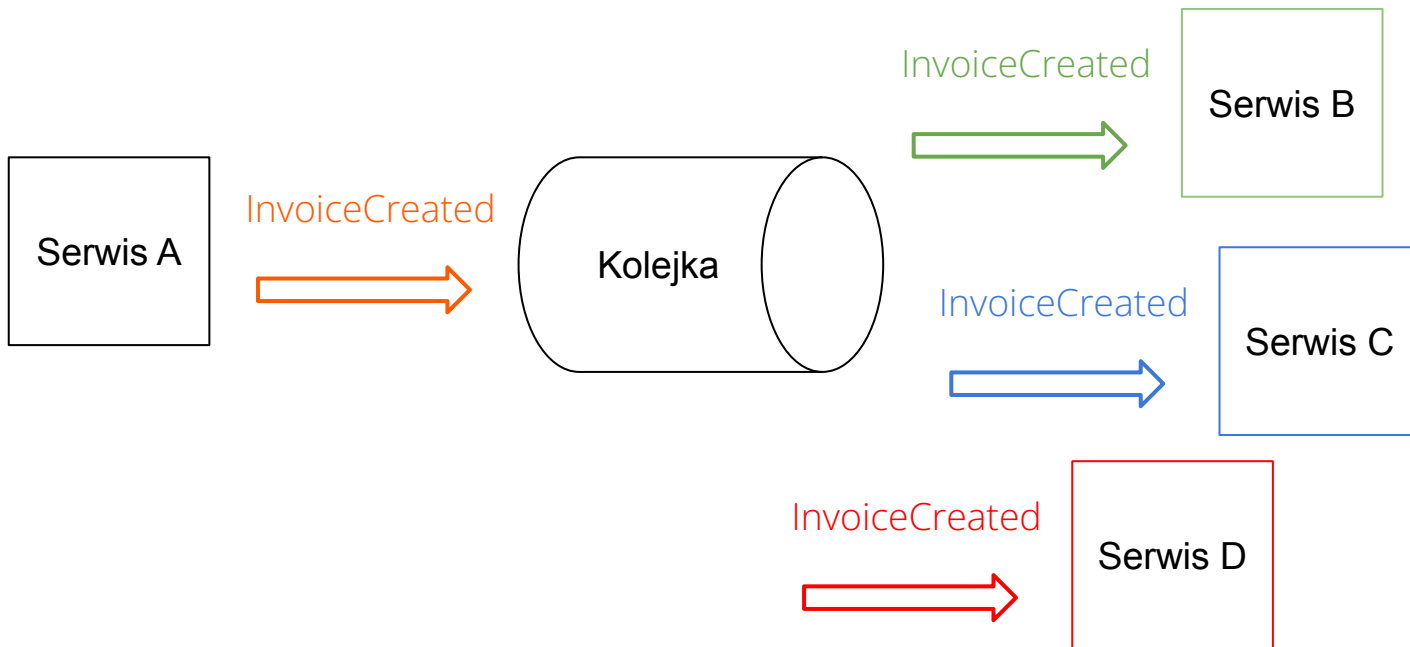
Naiwne podejście do powiadamiania innych usług

Serwis synchronicznie powiadamia inny serwis o zmianie w wyniku jakiejś operacji, tylko co się stanie jak więcej serwisów będzie chciało nasłuchiwać na tę zmianę? Problemy ze skalowaniem.



Pattern: Zdarzenia z wykorzystaniem pub-sub

W celu powiadomienia odbiorców o zdarzeniu które wydarzyło się w wyniku jakiejś operacji wykorzystywany jest mechanizm pub-sub (nadawca-odbiorca). W mechanizmach pub-sub typowo wielu odbiorców nasłuchuje na pojedyncze zdarzenie.



Kolejka wiadomości vs Pub-Sub

Kolejka wiadomości	Pub-Sub
Pojedyncza wiadomość trafia do jednego odbiorcy	Kopie pojedynczej wiadomości trafiają do wielu odbiorców
Wielu odbiorców będzie odbierało różne wiadomości	Wielu odbiorców będzie odbierało kopie tych samych wiadomości
Producent zleca jednostkę pracy i chce aby była wywołana zazwyczaj raz	Producent wysyła potencjalnie interesujące zdarzenie, nie dba kto je otrzyma

Komunikacja synchroniczna vs asynchroniczna

Komunikacja synchroniczna	Komunikacja asynchroniczna
Od razu znamy rezultat operacji	Na rezultat operacji musimy czekać
Użytkownik od razu widzi rezultat operacji	Użytkownik musi czekać na rezultat operacji
Otrzymujemy spójny stan, dane zmieniają się transakcyjnie	Stan jest ostatecznie spójny
Z reguły proces biznesowy wykonuje się szybciej (mniej IO)	Z reguły proces biznesowy wykonuje się wolniej (naruszenie kolejek, baz danych, więcej odpytywania)
Łatwiejsza w analizie systemu i błędów (przykład z produkcji)	Trudniejsza w analizie systemu i błędów

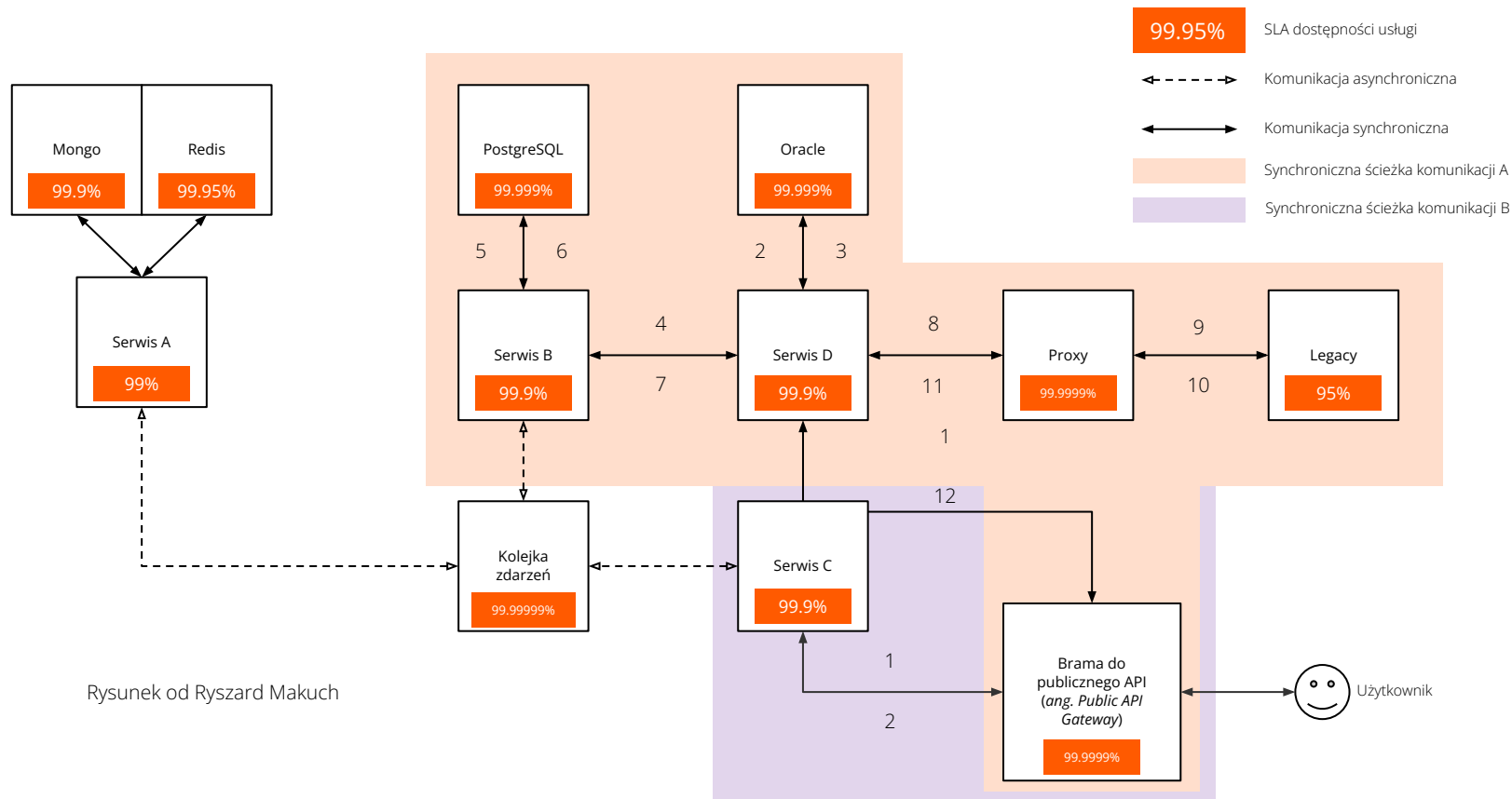
Komunikacja synchroniczna vs asynchroniczna

Komunikacja synchroniczna	Komunikacja asynchroniczna
Trudniej zaprojektować cykliczne zależności między serwisami, widać je od razu	Łatwiej zaprojektować cykliczne zależności między serwisami
Utrudnia skalowanie	Ułatwia skalowanie
Zwiększa powiązanie między serwisami (coupling)	Zmniejsza powiązanie między serwisami
Zmniejsza odporność na błędy	Zwiększa odporność na błędy
Zmniejsza SLA dostępności serwisów	Zwiększa SLA dostępności serwisów
Stawia usługi na ścieżce krytycznej	Usuwa usługi ze ścieżki krytycznej

Komunikacja synchroniczna vs asynchroniczna

Komunikacja synchroniczna	Komunikacja asynchroniczna
Dobrze sprawdza się w monolicie	Dobrze sprawdza się w monolicie
Jest nienaturalna dla mikroserwisów	Jest naturalna dla mikroserwisów
Utrudnia zbudowanie poprawnej architektury mikro-serwisowej	Ułatwia zbudowanie poprawnej architektury mikro-serwisowej
Wbrew pozorom, często nie odzwierciedla tego jak działa proces biznesowy	Wbrew pozorom, często lepiej odzwierciedla to jak działa proces biznesowy
Naturalna dla programisty, tak byliśmy uczeni programowania	Ciężka do zrozumienia dla programisty

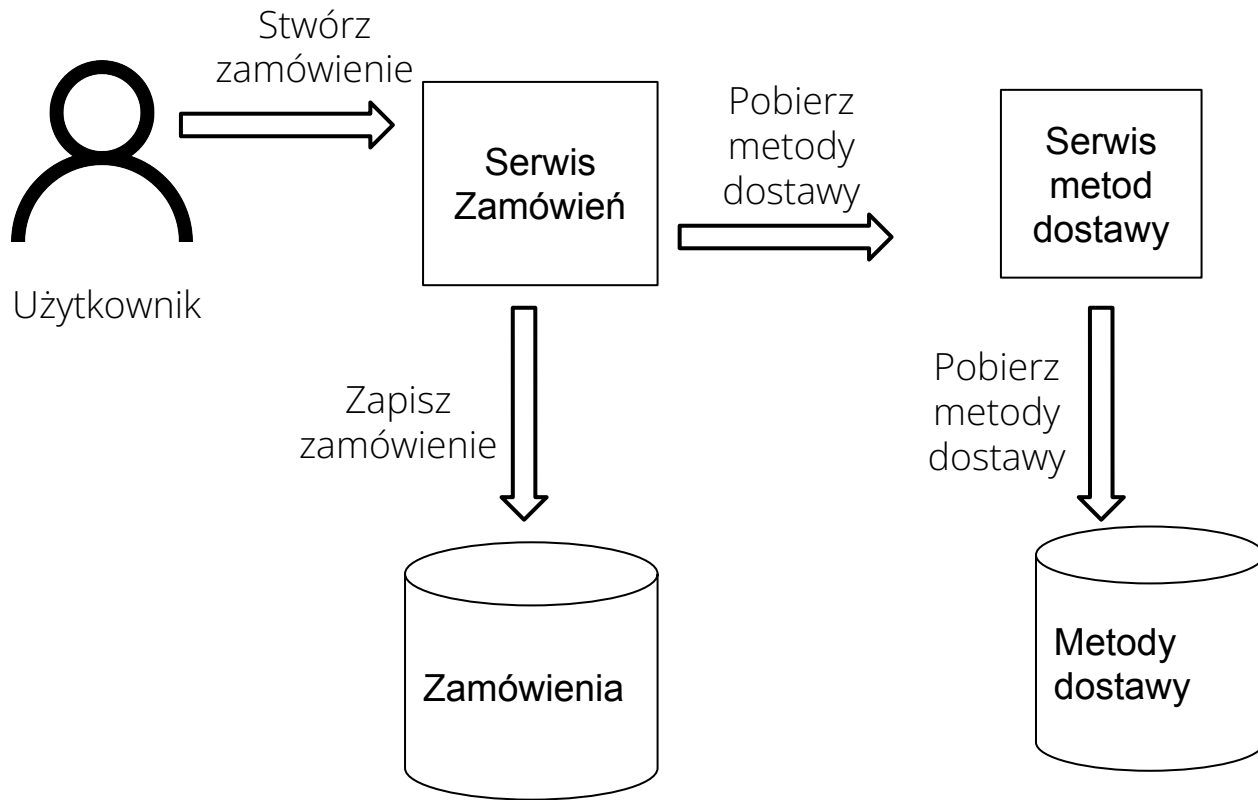
Jak zdarzenia zwiększają SLA aplikacji



Rysunek od Ryszard Makuch

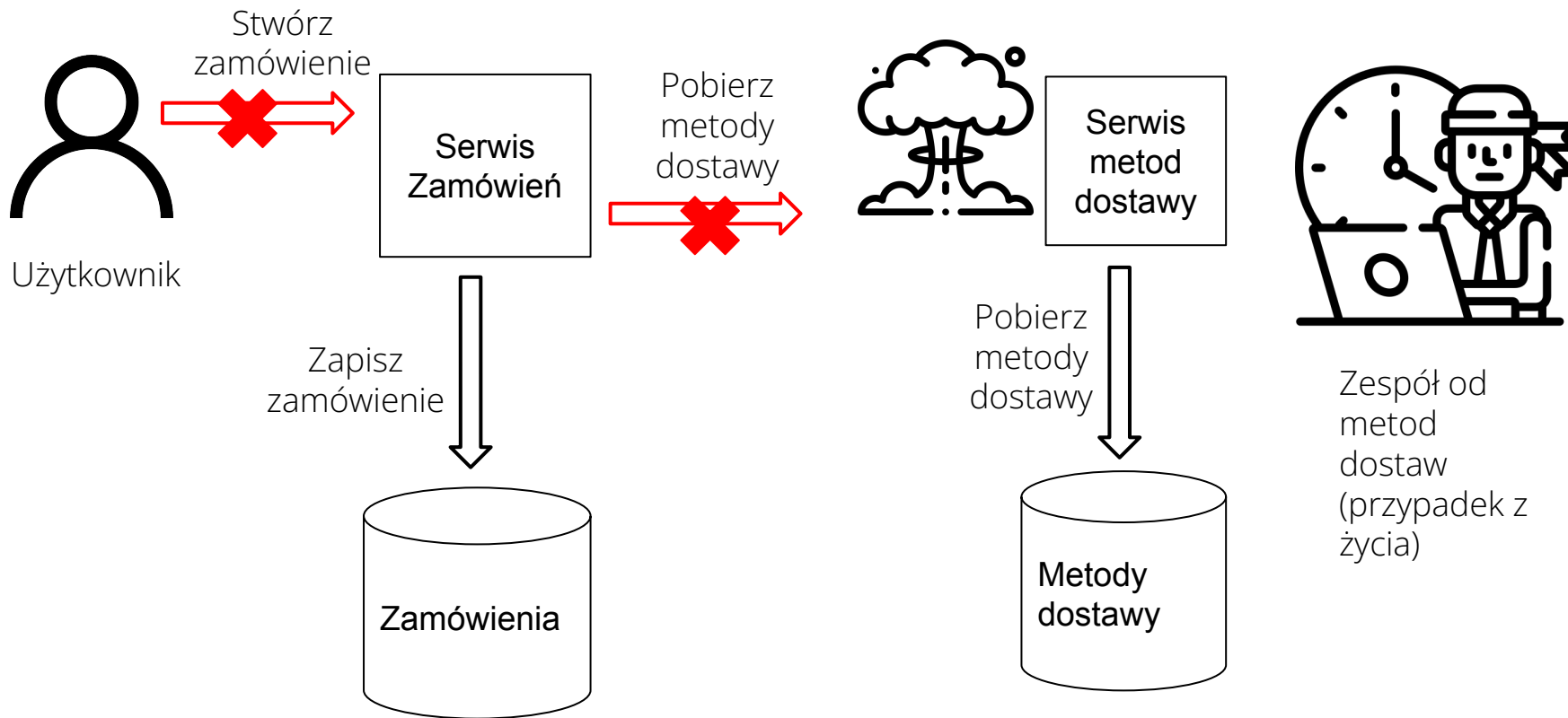
Architektura bez read-modeli - antypattern mikroserwisowy

Serwis zamówień odpytuje serwis metod dostaw o potrzebne metody dostawy, wszystko na pierwszy rzut oka zdaje się działać poprawnie



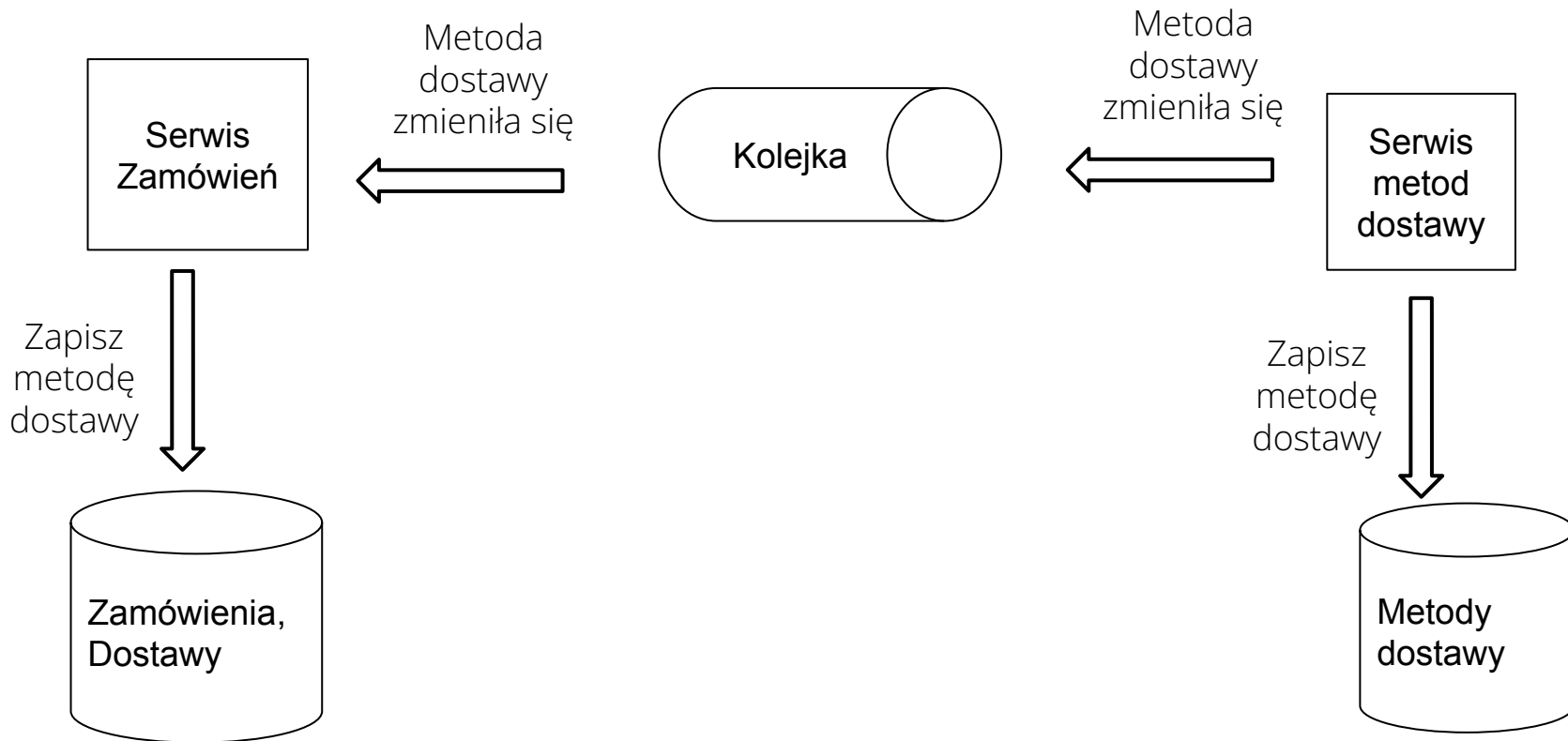
Architektura bez read-modeli - antypattern mikroserwisowy

Serwis metod dostawy ulega awarii, użytkownik nie może zrealizować zamówienia, zespół metod dostawy stara się naprawić błąd, czas ucieka. Zespół **boi się** wdrażać nowe zmiany.



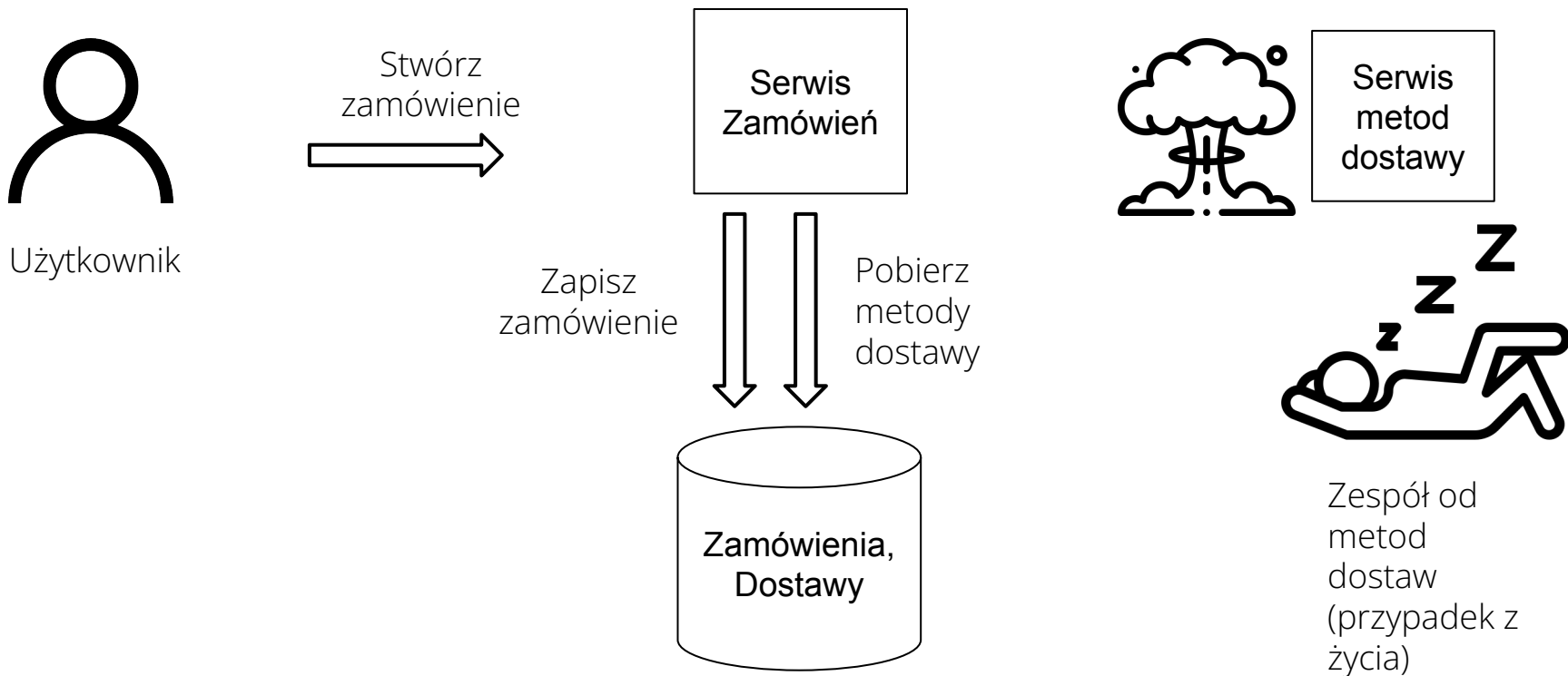
Pattern: Architektura z wykorzystaniem read modeli

Serwis zamówień posiada prywatny read model który jest aktualizowany zdarzeniami publikowanymi przez serwis metod dostawy



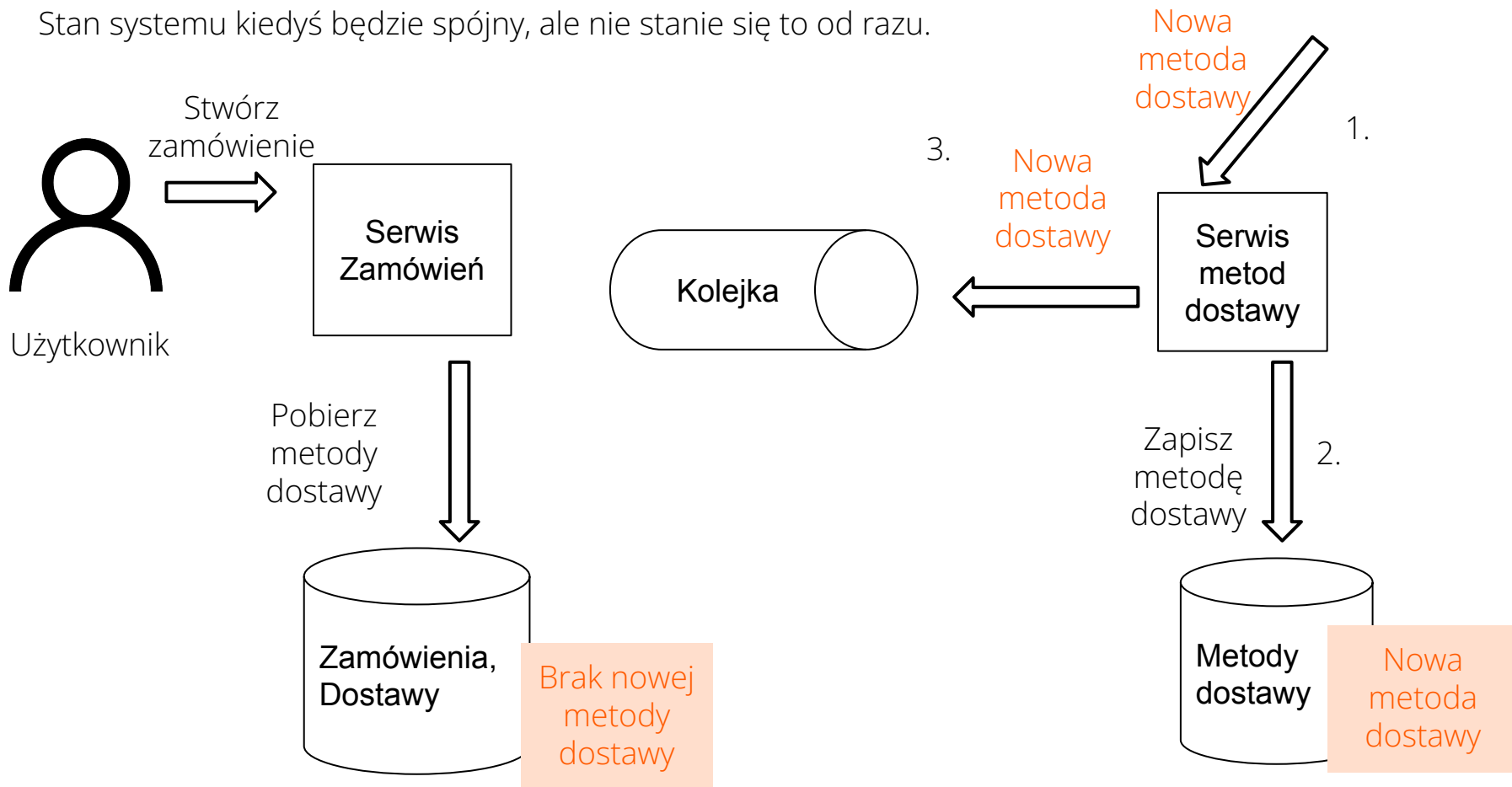
Pattern: Architektura z wykorzystaniem read modeli

Serwis metod dostawy ulega awarii, serwis zamówień nawet o tym się nie dowiaduje ponieważ nie jest mocno powiązany z serwisem metod dostawy. Zespół metod dostaw ma czas na naprawę awarii, ponieważ nie jest na **ścieżce krytycznej**. Zespół jest **wyspany** i **nie boi** się wdrażać.



Ostateczna spójność (eventual consistency)

Stan systemu kiedyś będzie spójny, ale nie stanie się to od razu.



Typowe problemy w komunikacji asynchronicznej

Ostateczna spójność (eventual consistency)

- Stan systemu **kiedyś** będzie spójny, ale nie stanie się to od razu
- Przez pewien czas część systemu nie będzie miała informacji którą ma inna część systemu
- Czasem jest to duży problem dla biznesu
- Ale najczęściej nie jest to problem ponieważ biznes często działa asynchronicznie

Przykłady asynchronicznych i ostatecznie spójnych procesów biznesowych

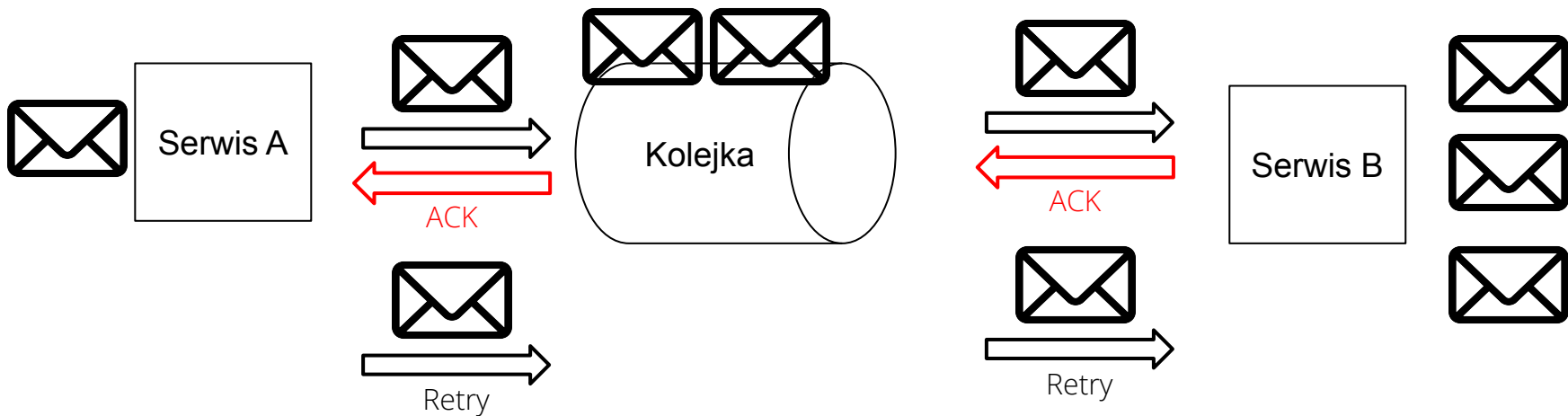
- Ręczna weryfikacja reklamacji (pracownik musi usiąść i potwierdzić).
- Przelewy
- Opłaty
- Wystawienie faktury
- Wstawienie postu na facebook
- Rezerwacja hotelu
- Wysłanie przesyłki

Gwarancja “Exactly once”

Wyróżniamy 3 rodzaje gwarancji dostarczania wiadomości

- At most once - wiadomość zostanie dostarczona jeden raz lub zostanie zgubiona i nie pojawi się nigdy
- At least once - wiadomość zostanie dostarczona jeden raz lub dwa lub trzy lub $\backslash_(_)_/_$
- Exactly once - wiadomość zostanie dostarczona dokładnie jeden raz, wiele kolejek lubi chwalić się że zapewnia “exactly once”, ale warto przeczytać co napisali małym druczkiem :)

Jeżeli mamy IO to gwarancja exactly once jest niemożliwa do osiągnięcia, zawsze może wydarzyć się coś co sprawi że potwierdzenie odebrania wiadomości zostanie zgubione. Możliwe powody: timeout, śmierć instancji, ...



Deduplikacja i zachowanie kolejności

Jeżeli chcemy przetwarzać wiadomości dokładnie raz:

- Producent musi nadawać wiadomościom unikalne identyfikatory.
- Konsument musi transakcyjnie zapisywać (na przykład w bazie danych) przetworzone wiadomości razem z rezultatami ich przetworzenia. (Na przykład wiadomość nakazująca stworzyć fakturę i stworzona w rezultacie faktura).

Niektóre kolejki nie zapewniają dostarczania wiadomości w kolejności ich wysłania.

- Producent musi nadawać wiadomościom timestampy lub (jeszcze lepiej) monotonicznie rosnące numery/wersje.
- Konsument czytając eventy musi brać pod uwagę timestamp i odpowiednio aplikować wiadomość.

Dwa podejścia do zdarzeń

Zdarzenia przyrostowe (event sourcing)

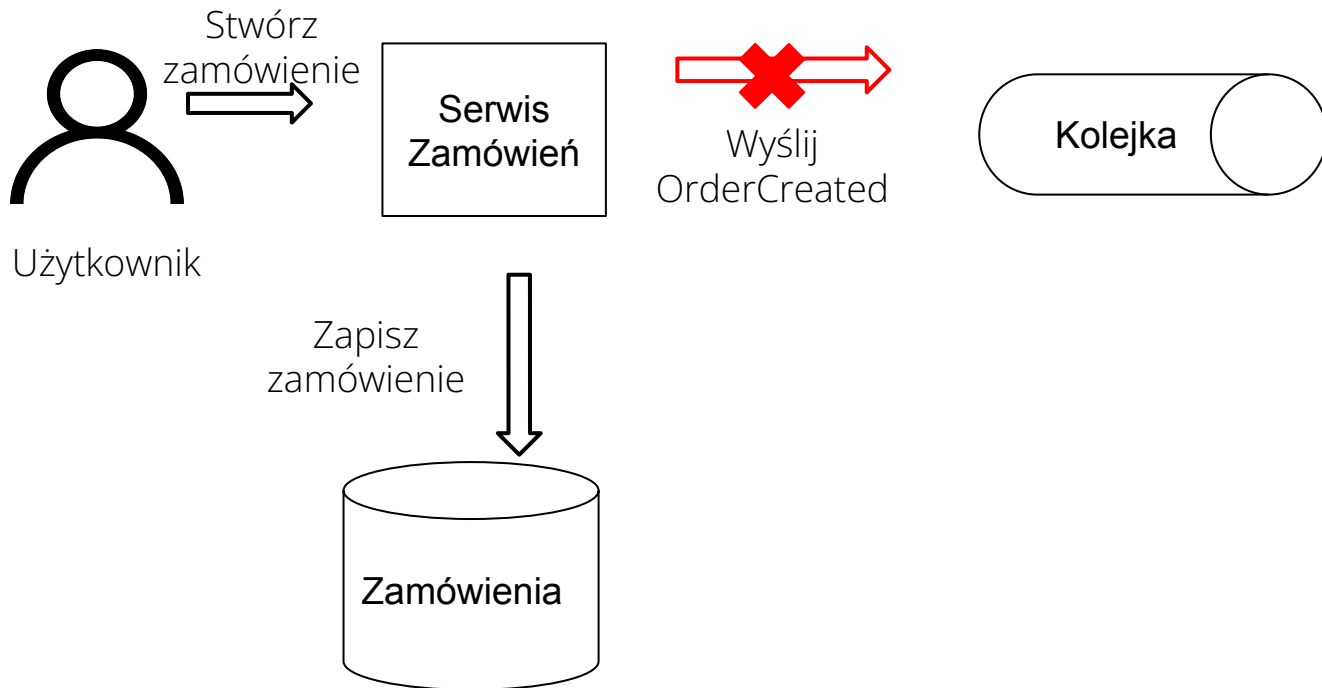
- Zdarzenia opisują pewną historię zmian
- Zawierają jedynie zmianę danego pola czy zestawu pól
- ***OfferCreated, OfferNameChanged, OfferPriceChanged, OfferEnded.***
- ***OfferPriceChanged (price: 100 -> 120), OfferNameChanged (name: "Lody" -> "Chipsy")***
- Tego typu zdarzenia oferują większą granularność ale są trudne do zachowania kolejności (musimy brać pod uwagę starsze zdarzenia ponieważ mogą zawierać cenną zmianę stanu).

Zdarzenia snapshotowe

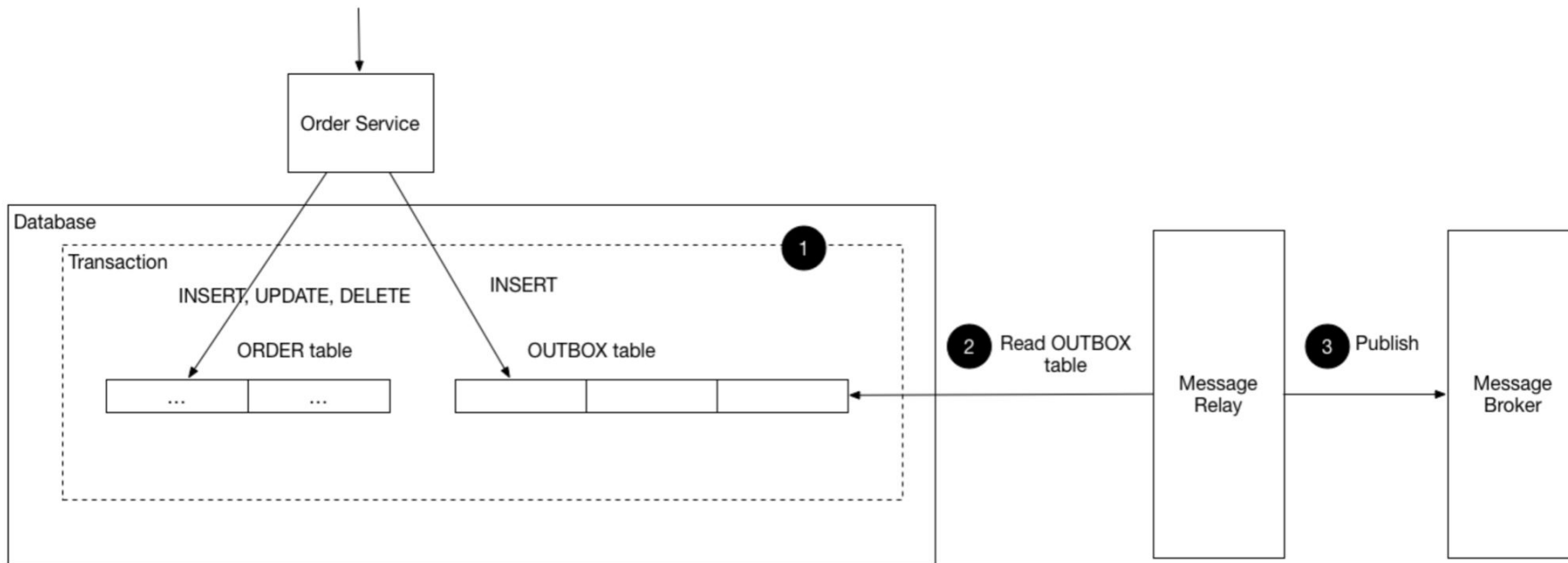
- Zdarzenia opisują cały stan po zmianie
- Zwracany jest podobny model jak w przypadku zasobu RESTowego
- ***OfferChanged(price:120, name : "Lody"), OfferChanged(price:120, name : "Chipsy")***
- Tego typu zdarzenia są mniej "zgodne ze sztuką", za to łatwe w zachowaniu kolejności (możemy ignorować starsze wiadomości ponieważ wiemy że widzieliśmy już nowszy stan).

Transakcyjność

Jak zapisać dane do dwóch systemów na raz?



Transactional Outbox



<https://microservices.io/patterns/data/transactional-outbox.html>

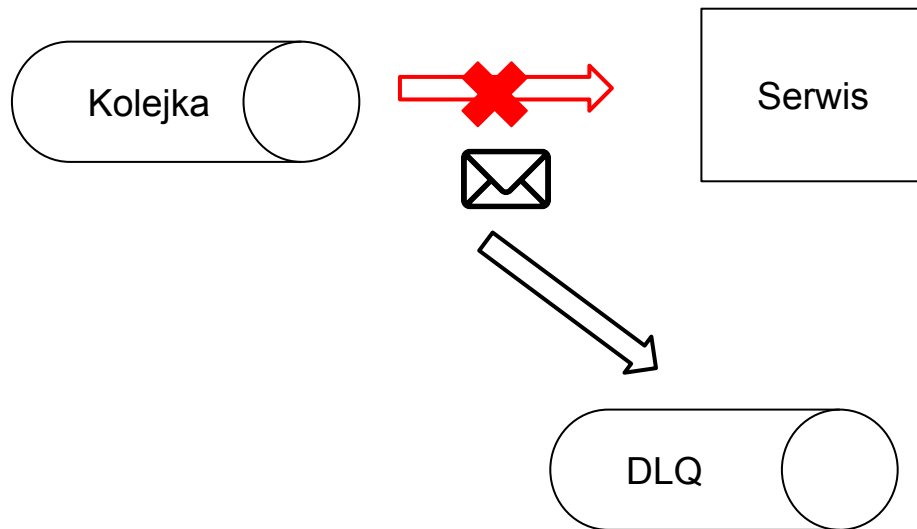
Transactional Outbox w Mongo?

Granicą transakcji jest dokument, wiadomości trzeba zapisywać razem z danymi biznesowymi w dokumencie

```
{
  "orderId": "a03427bd-cdd7-4f75-b495-99442c904dd9"
  "productId": "6d7781ac-0354-43ea-a563-8b8ac139b01a",
  ...
  "outboxMessages": [
    {"messageId": "90a772c3-e551-4444-8e96-8e193296df07, ... },
    {"messageId": ... },
    {"messageId": ... }
  ]
}
```


DLQ - Dead Letter Queue

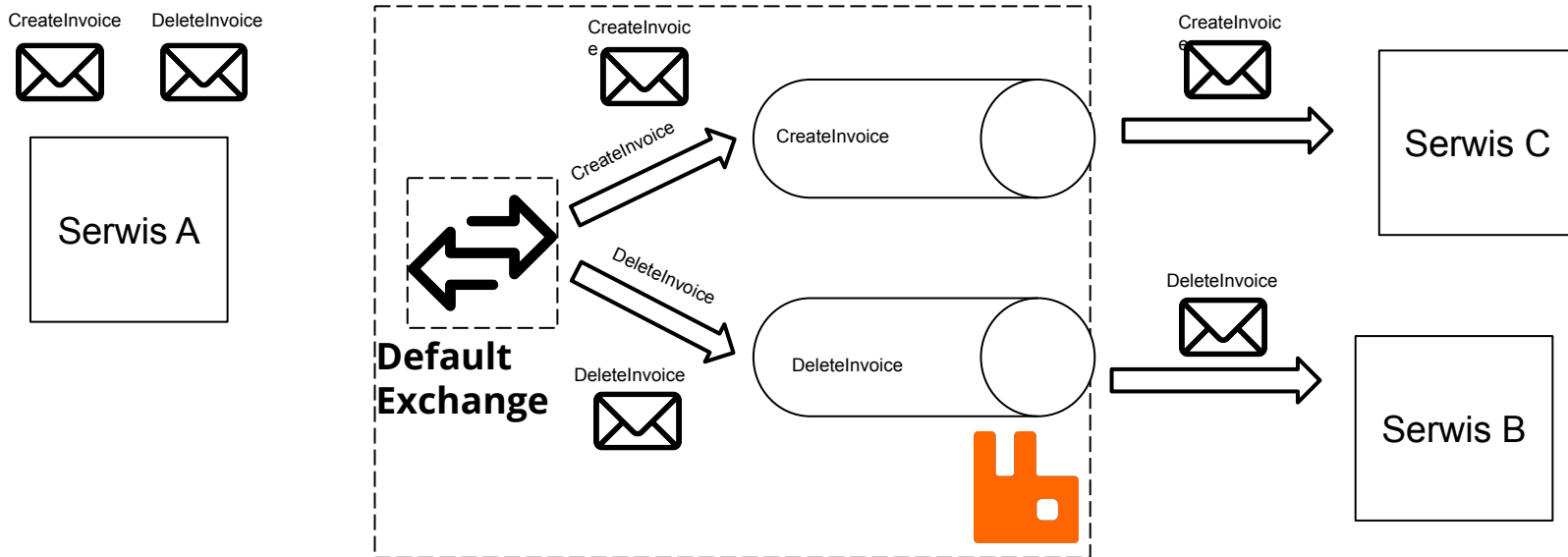
- Czasem wiadomość nie może zostać odebrana przez konsumenta (na przykład przez zwykły `NullPointerException` w kodzie).
- **Poison Message** - wiadomość która przekroczyła maksymalną ilość ponowień i blokuje subskrypcję.
- **Dead Letter Queue** - specjalna kolejka która służy do przechowywania niedostarczonych wiadomości



Rabbit

Rabbit

- Kolejka wiadomości - składa się z 2 rodzajów elementów zamkniętych w jednej aplikacji - exchange oraz kolejek.
- Producent wysyła wiadomości na exchange
- Exchange routuje wiadomości do innych exchange albo kolejek za pomocą **bindingów**.
- **Wiele** kolejek lub exchange może być podpiętych do **wielu** exchange, możemy konstruować skomplikowane routingi.
- Przykład **Default Exchange**: wiadomości lecą do kolejek których **nazwa** odpowiada **kluczom** wiadomości.
- Default Exchange jest tworzony zawsze, tworzy bindingi których klucze odpowiadają nazwom kolejek.

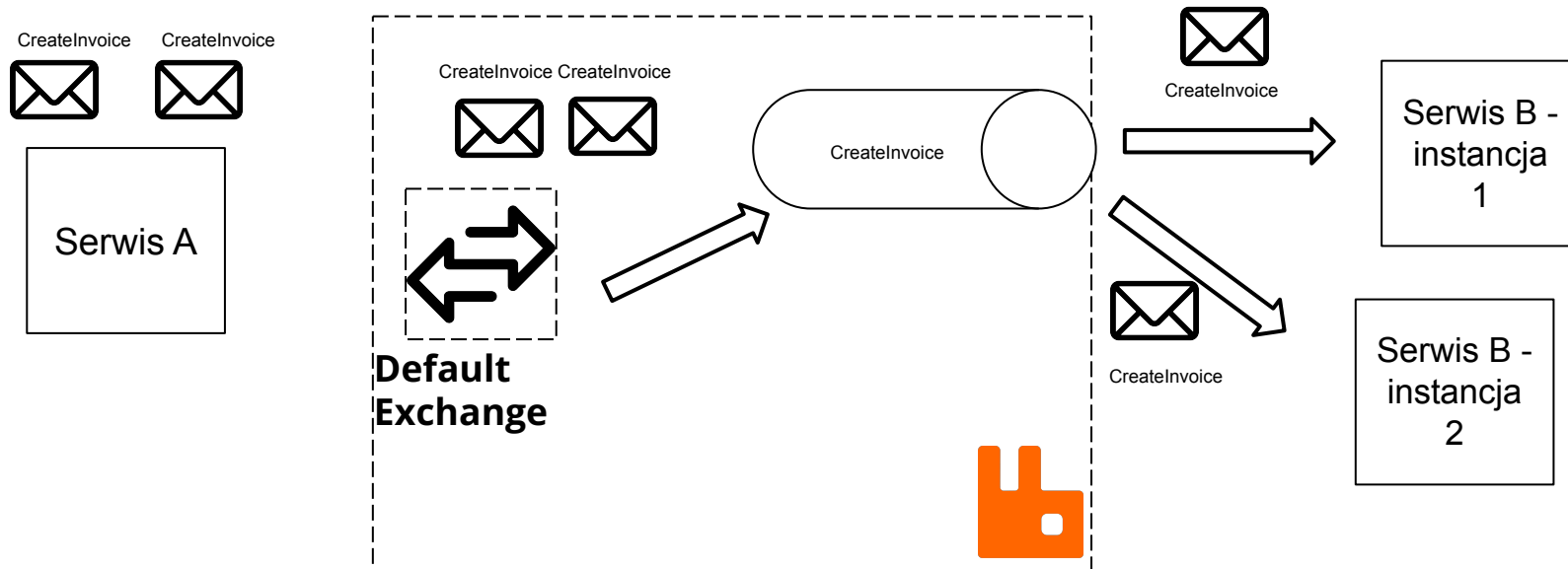


Rabbit

- Rabbit zapewnia zachowanie kolejności wiadomości od jej wysłania przez nadawcę na exchange aż odebrania przez konsumenta z kolejki.
- Jeżeli mamy wiele instancji konsumentów z tej samej kolejki to nie zachowamy kolejności.
- Rabbit **usuwa** wiadomość po potwierdzeniu odebrania przez konsumenta.

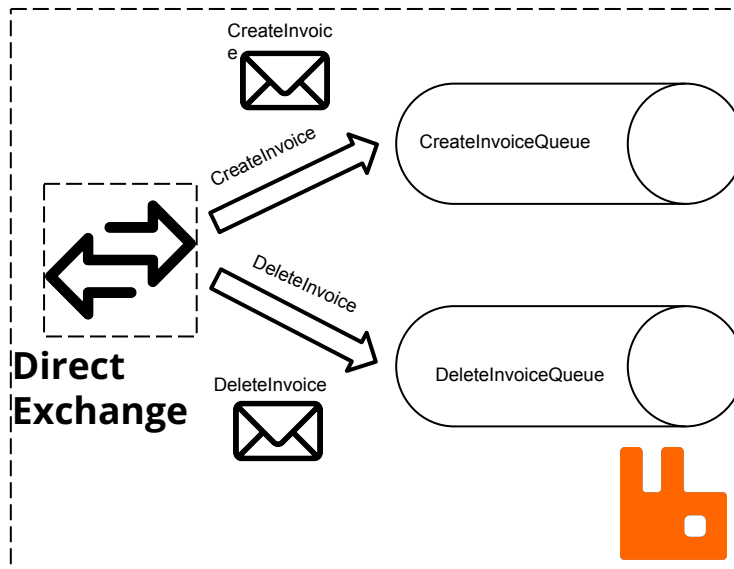
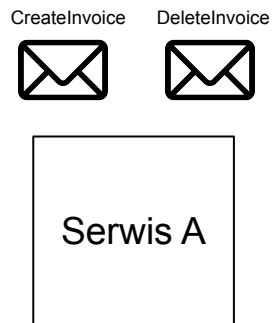
Rabbit

W przypadku wielu subskrybujących kolejka wysyła wiadomości w systemie round robin, po kolei do każdego



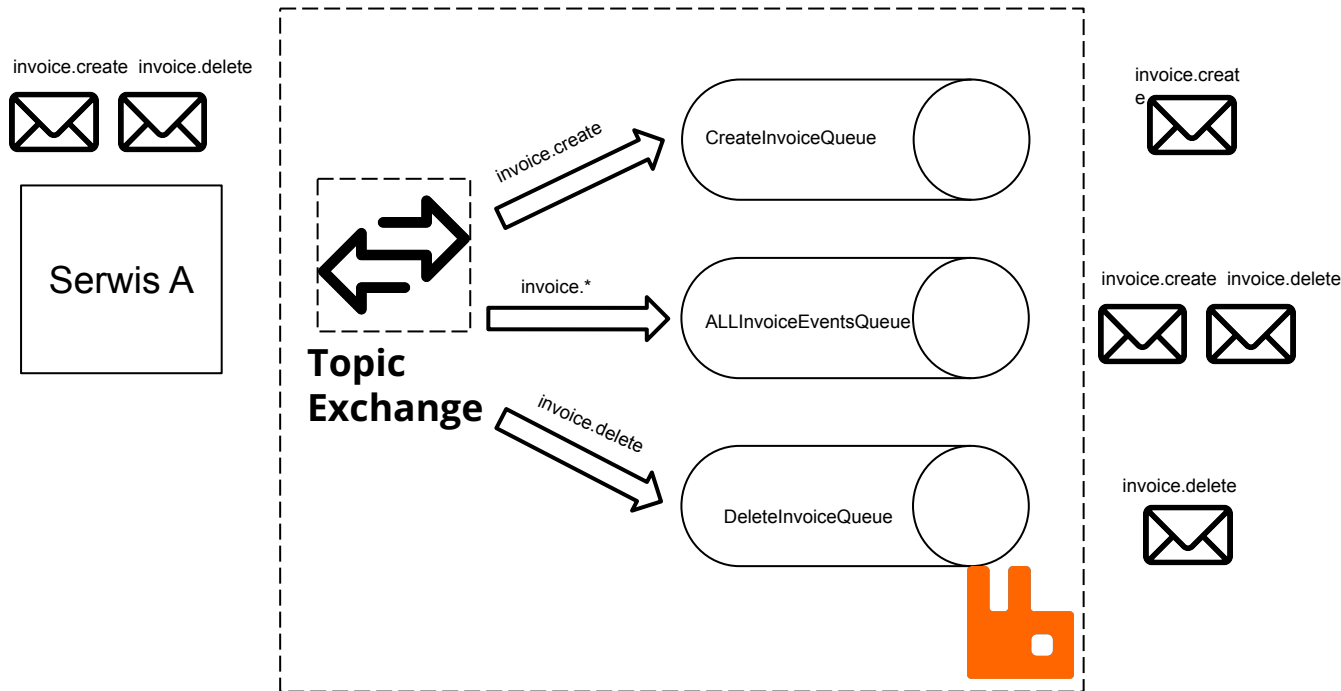
Rabbit

Direct Exchange: wiadomości lecą do kolejek których **binding key** odpowiada **kluczom wiadomości**



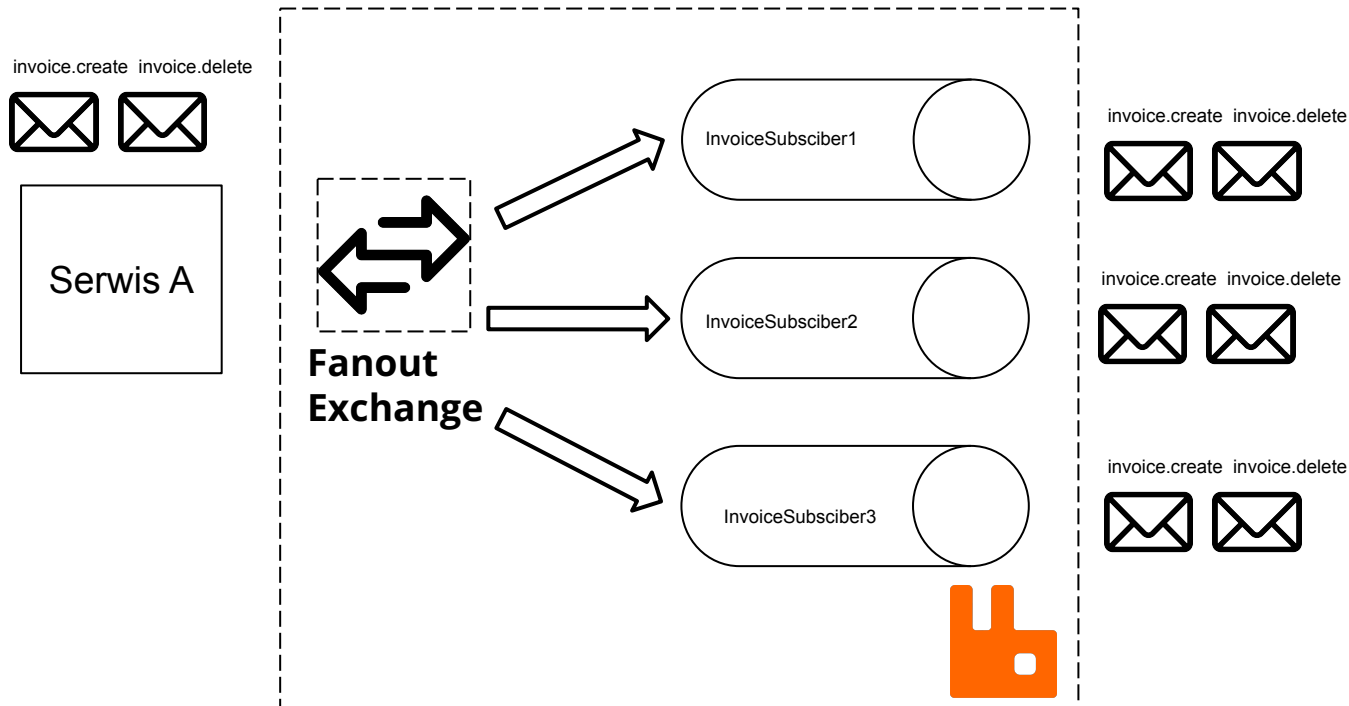
Rabbit

- **Topic Exchange:** wiadomości lecą do kolejek których **binding key pattern** odpowiada **kluczom wiadomości**
- Bardzo podobny do direct exchange różnica polega na tym że dopasowujemy pattern zamiast dokładnego klucza.



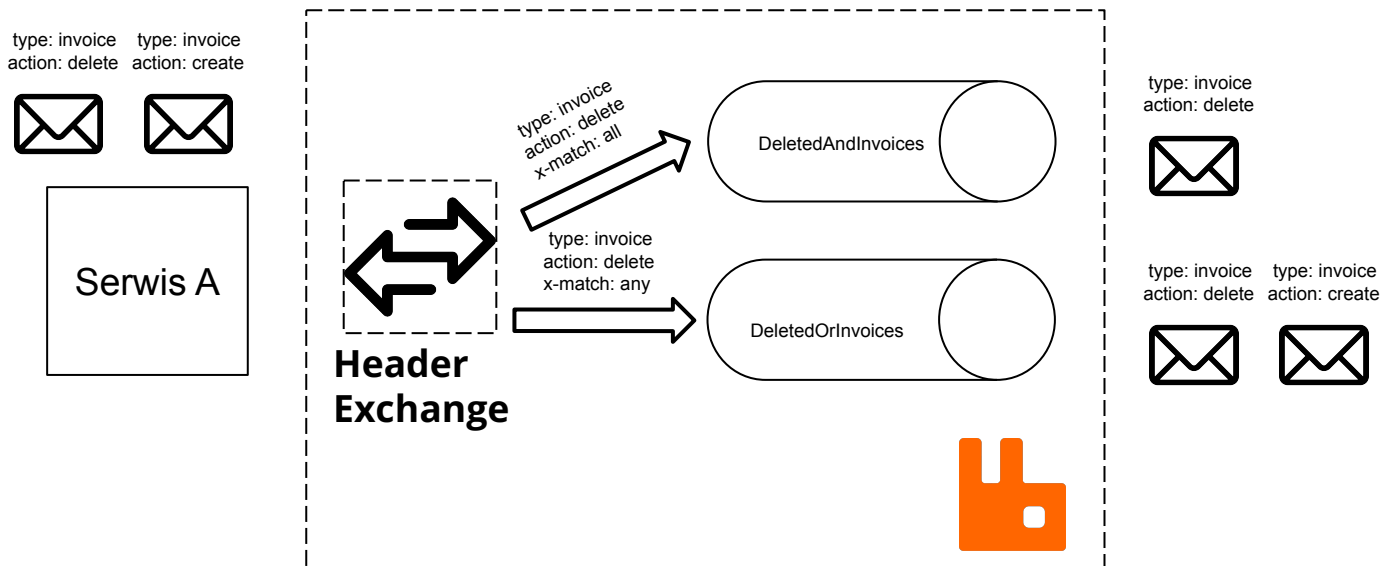
Rabbit

- **Fanout Exchange**: wiadomości lecą do **wszystkich** kolejek bez zwracania uwagi na binding key.
- Najbardziej przypomina model kafki (pub-sub).



Rabbit

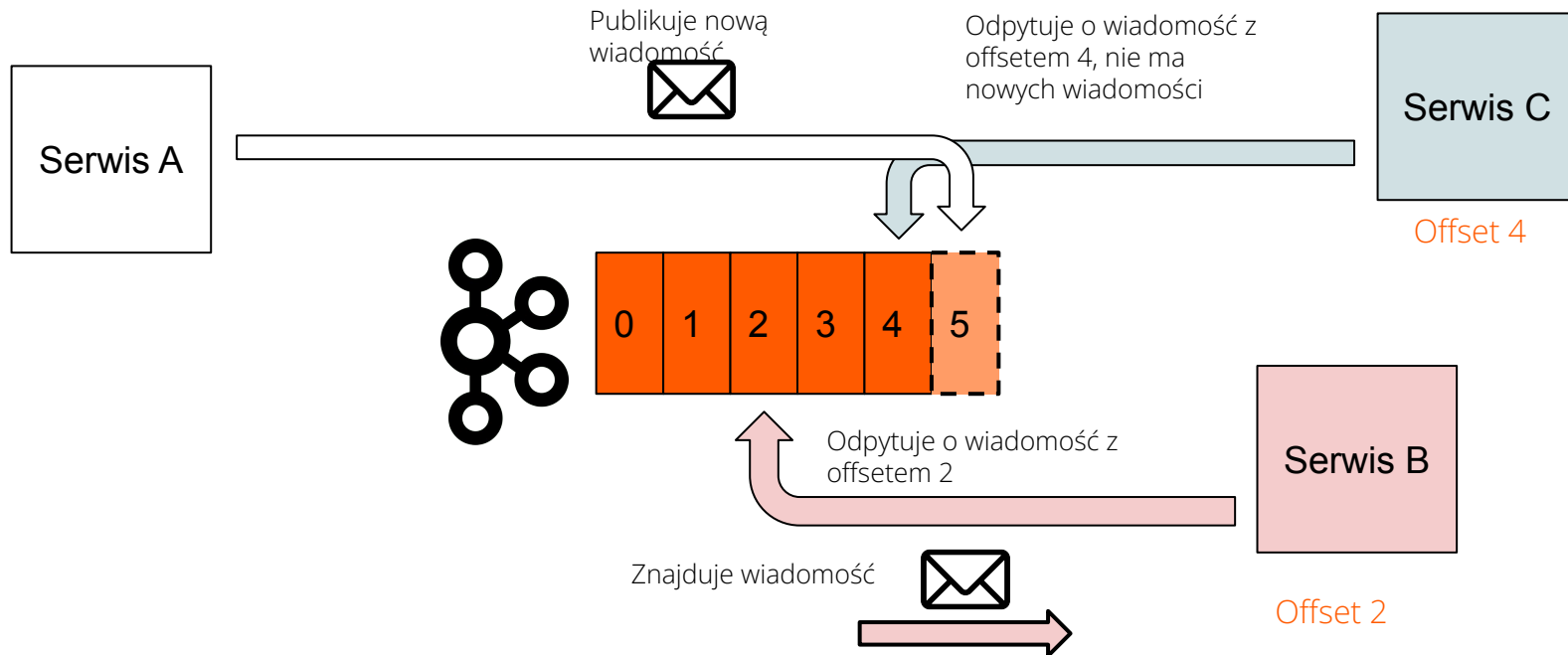
- **Header Exchange:** wiadomości lecą do kolejek których **binding arguments** odpowiadają **headerom wiadomości**
- Podobny do topic exchange różnica polega na tym że dopasowujemy dokładne wartości headerów zamiast patternów kluczy.
- Możemy sprecyzować czy chcemy dopasować jakikolwiek header (**any**) czy wszystkie (**all**) za pomocą argumentu **x-match**.



Kafka

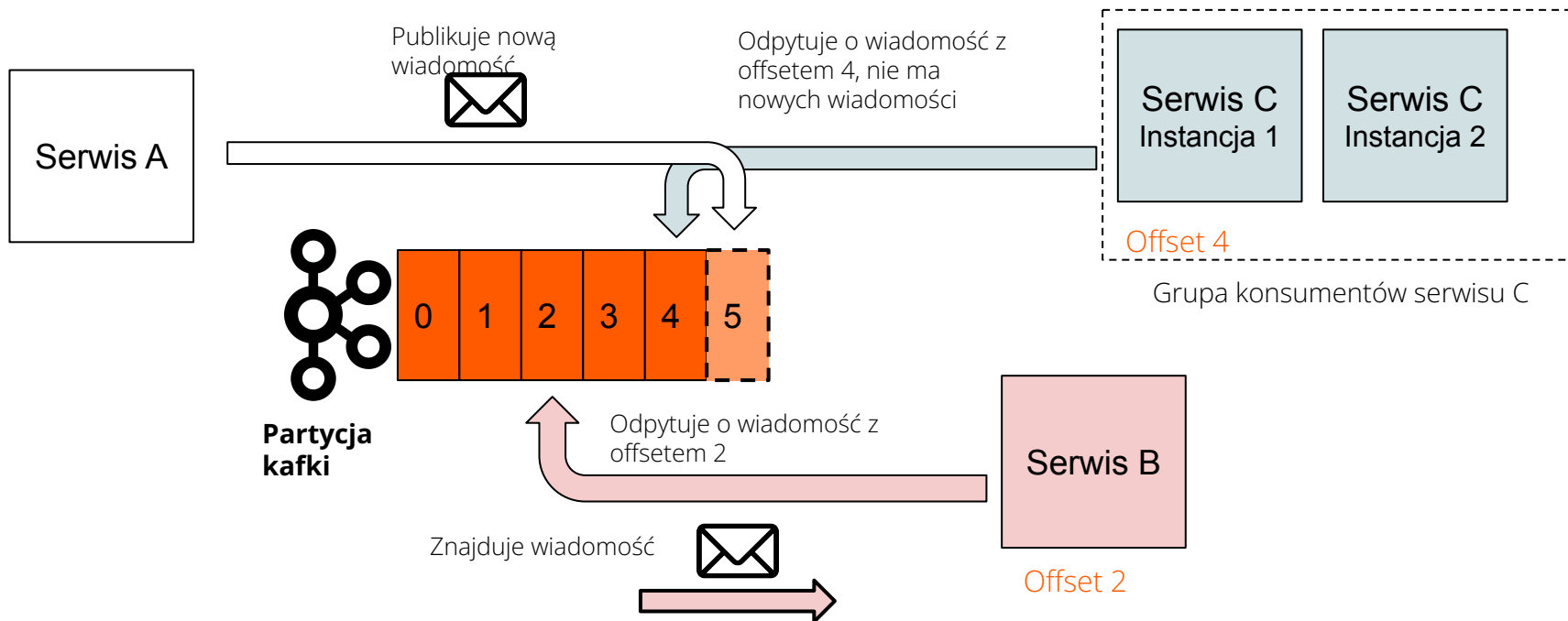
Kafka

- Log wiadomości - producent publikuje wiadomości, konsumenci odpytują cyklicznie kafkę o nowe zdarzenia w mechanizmie pull.
- Konsumenci zapisują offsety na kafce z własnymi interwałami.
- Mechanizm pull pozwala na łatwe rozszerzanie ilości konsumentów.
- Kafka **nie usuwa** wiadomości po odczytaniu przez konsumenta.



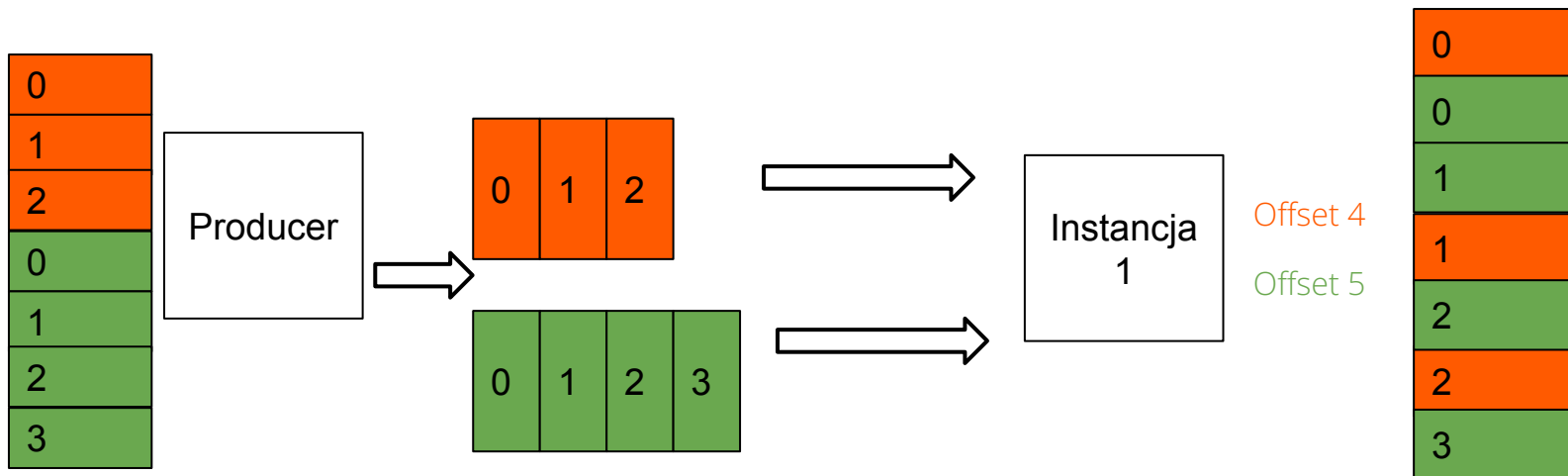
Kafka

Grupy konsumentów - instancje przypisywane są do grup konsumentów, każda grupa ma swoje własne offsety, z jednej partycji może czytać tylko jeden konsument z grupy. Partycja gwarantuje dzięki temu kolejność wiadomości.



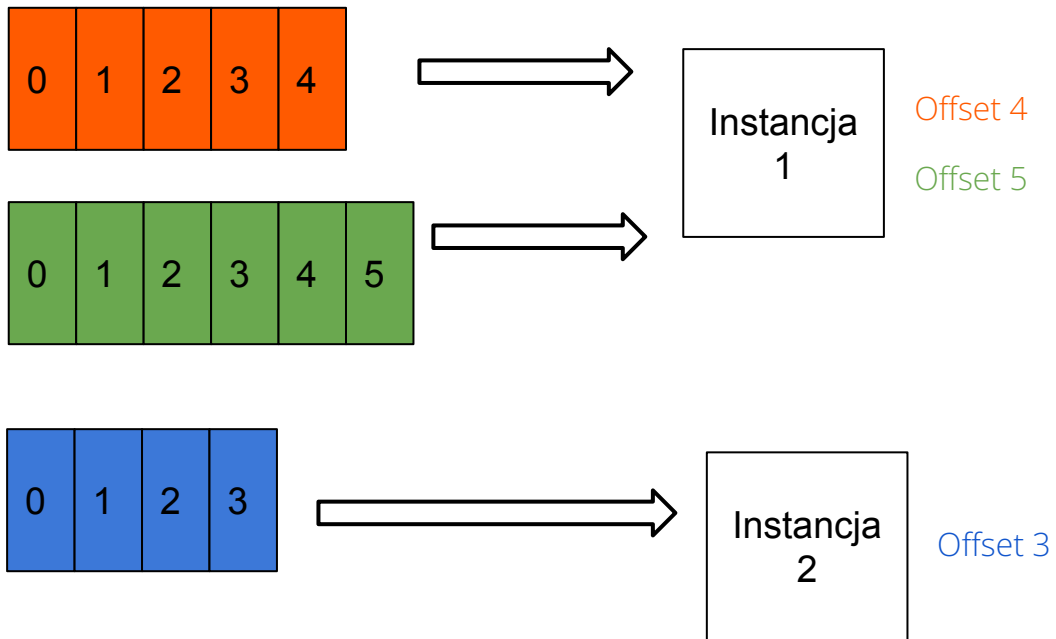
Kafka - partycje

- W obrębie jednej partycji mamy zapewnioną kolejność eventów, uznajemy że jeden konsument to tak jakby jeden wątek.
- Między partycjami kolejność nie jest zachowana.
- Wiele partycji pozwala na dużo lepszą skalowalność (wiele równoległych odczytów)
- Wiadomości przypisywane są do partycji losowo lub poprzez klucz partycjonujący (jeżeli chcemy aby odpowiednie wiadomości trafiły na te same partycje



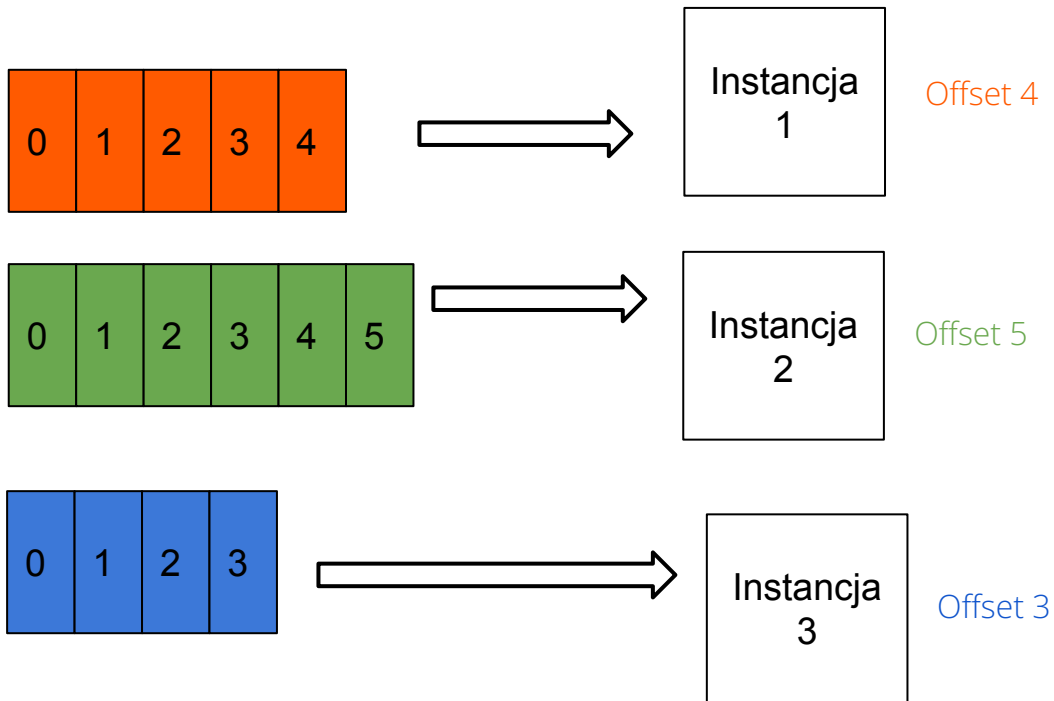
Kafka - partycje

Więcej partycji niż instancji w grupie konsumentów



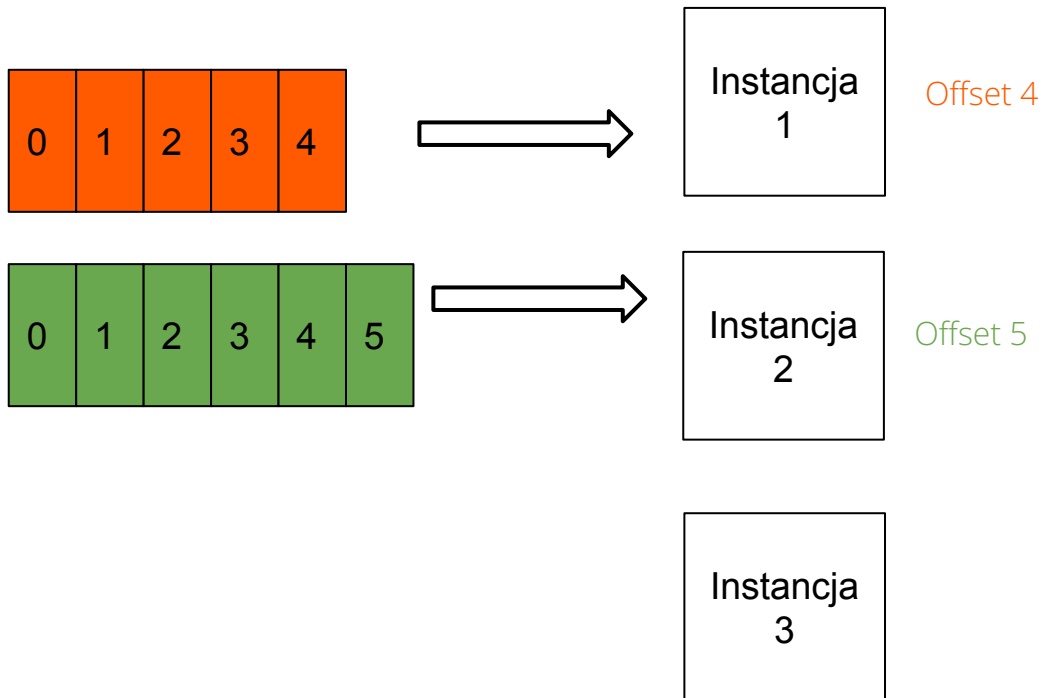
Kafka - partycje

Tyle samo partycji co instancji w grupie konsumentów



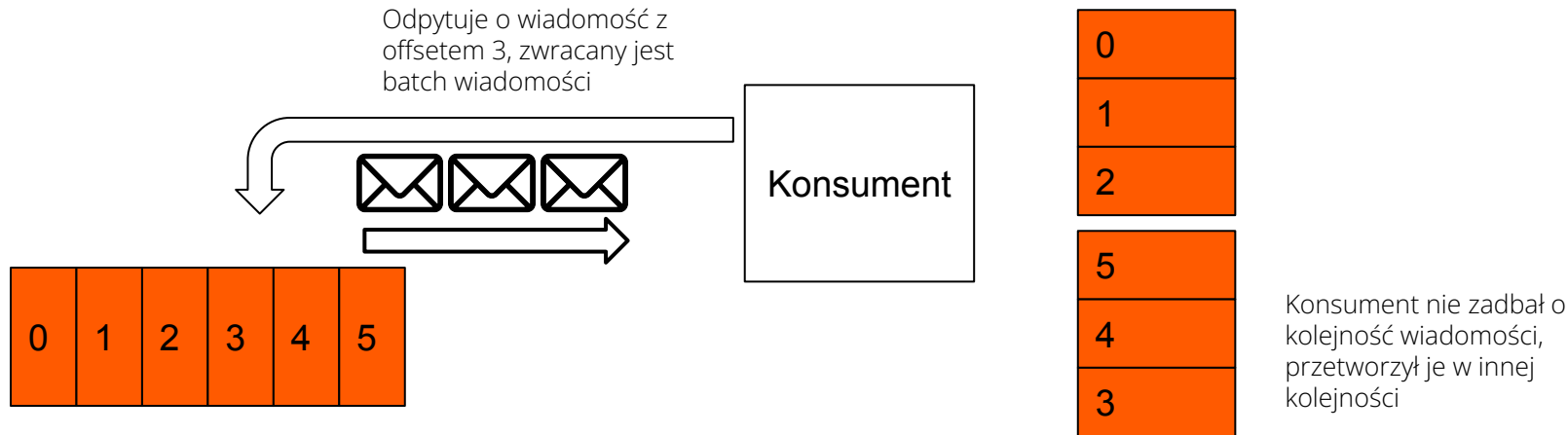
Kafka - partycje

Mniej partycji niż instancji w grupie konsumentów



Kafka - przetwarzanie batchowe

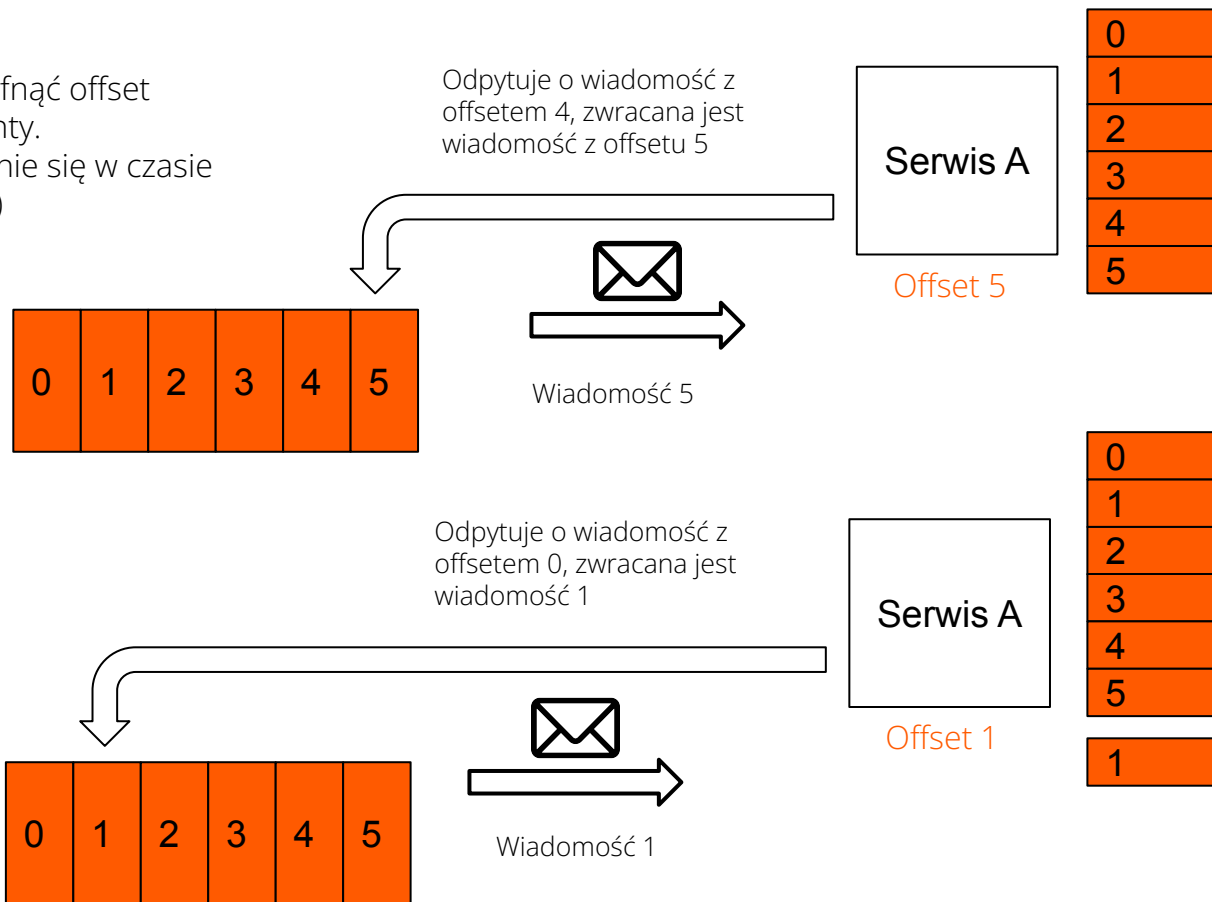
- Jedna instancja przetwarza z jednej partycji sekwencyjnie (jeden wątek), jest to dosyć wolne
- Dodatkowym podejściem do skalowania jest przetwarzanie batchowe, konsument w jednym zapytaniu pobiera wiele wiadomości które przetwarza wielowątkowo
- Konsument musi zadbać aby jego wątki dbały o kolejność (jeżeli jest ona istotna)



Kafka

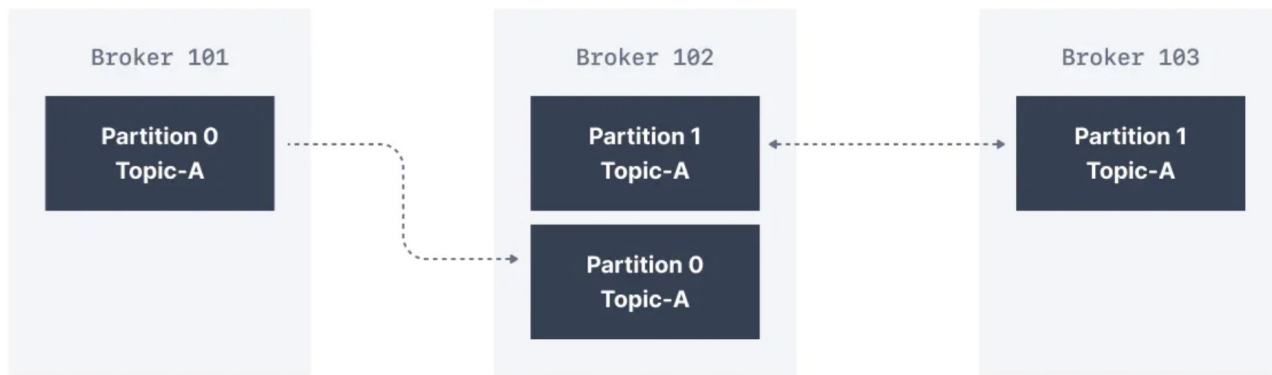
Cofanie offsetu - serwis może cofnąć offset aby jeszcze raz przetworzyć eventy.

Mechanizm ten pozwala na cofanie się w czasie aby na przykład poprawić błąd :)



Kafka - replikacja

W celu zachowania wysokiej dostępności (High Availability) kafka replikuje partycje pomiędzy węzłami klastra w modelu master-slave.



Kafka Topic Replication

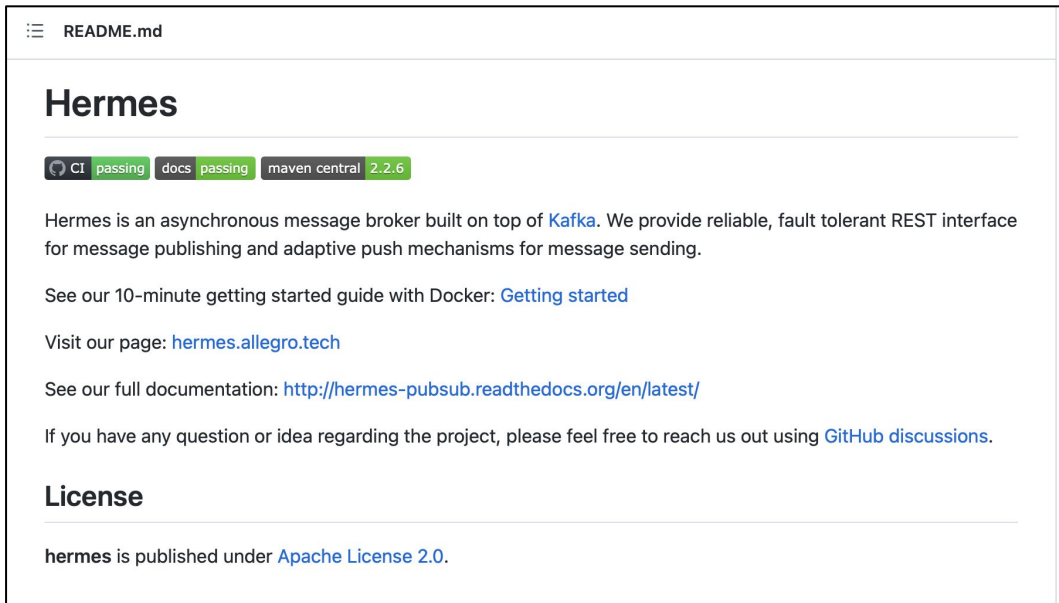
Kolejka wiadomości vs Pub-Sub

Cecha	Rabbit	Kafka
Mechanizm	Push - Smart	Pull - Dumb
Skalowanie	Wertykalne - dokładanie ramu	Horyzontalne - dodanie maszyny do klastra
Długość życia wiadomości	Usuwa wiadomość po odebraniu	Trzyma wiadomość tak długo jak retencja pozwala
Kolejność wiadomości	W obrębie kolejki (queue)	W obrębie partycji
Replay wiadomości	Nie	Tak długo jak retencja pozwala
Routowanie i filtrowanie wiadomości	Możliwe do odpowiednich klientów przez odpowiednie kolejki	Nie możliwe, wszyscy czytają cały topic
Wysoka dostępność	Tak, poprzez replikację na inne maszyny w clustrze	Tak, poprzez replikację na inne maszyny w clustrze

Hermes

Hermes

- Allegrowy projekt open-source
- Główna kolejka wiadomości w allegro
- Działa w oparciu o kafka
- Dodaje komunikacje przez REST
- Zachowuje kafkowe topicki
- Mechanizm push na subskrybcje
- link : <https://github.com/allegro/hermes>

A screenshot of the Hermes project's README file. The page has a light blue header with a hamburger menu icon and the text 'README.md'. The main title 'Hermes' is in a large, bold, black font. Below the title, there are three status badges: 'CI passing' (green), 'docs passing' (green), and 'maven central 2.2.6' (green). The text describes Hermes as an asynchronous message broker built on top of Kafka, providing a reliable, fault-tolerant REST interface for message publishing and adaptive push mechanisms. It includes links to a 'Getting started' guide, the project page 'hermes.allegro.tech', and full documentation at 'http://hermes-pubsub.readthedocs.org/en/latest/'. It also mentions that users can reach out via 'GitHub discussions'. A 'License' section at the bottom states that Hermes is published under the 'Apache License 2.0'.

☰ README.md

Hermes

CI passing docs passing maven central 2.2.6

Hermes is an asynchronous message broker built on top of [Kafka](#). We provide reliable, fault tolerant REST interface for message publishing and adaptive push mechanisms for message sending.

See our 10-minute getting started guide with Docker: [Getting started](#)

Visit our page: hermes.allegro.tech

See our full documentation: <http://hermes-pubsub.readthedocs.org/en/latest/>

If you have any question or idea regarding the project, please feel free to reach us out using [GitHub discussions](#).

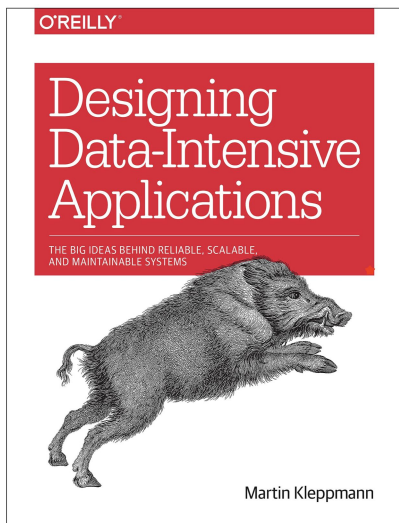
License

hermes is published under [Apache License 2.0](#).

Literatura

Literatura warta polecenia

- Designing Data-Intensive Applications - Martin Kleppmann
- Microservice Patterns - Chris Richardson



Warsztat

Czat do rozmowy w labie

- Kod źródłowy
 - <https://github.com/allegro-agh-2022/lab-kafka.git>
- Założenia
 - Kafka stoi na moim serwerze
 - Jeden wspólny topic: chat, 2 partycje
 - Każda grupa odpala aplikację na własnych serwerach
 - Każda aplikacja jest w we własnej consumer grupie
 - Aplikacja czyta wiadomości z dwóch partycji, zapisuje je w pamięci za pomocą biblioteki Reactor
 - Aplikacja wystawia endpointy do nasłuchiwania i wysyłania wiadomości

Czat do rozmowy w labie

- Obsługa aplikacji
 - Wysyłanie wiadomości: `curl 'localhost:8080/chat/send/{message}'`
 - Subskrybowanie wiadomości (SSE): `curl 'localhost:8080/chat/read?sender={CONSUMER_GROUP}'`
- Zadania
 - Zmodyfikuj metodę: `ChatEndpoint.read(String sender)` tak aby filtrowała wiadomości od danego sendera
 - Zmodyfikuj metodę: `ChatWriter.sendMessage()` tak aby wysyłać klucz partycjonujący
 - Odkomentuj `ChatListener.onParitionAssigned()` aby zresetować offset

Pytania?

michal.starosta@allegro.pl

Dziękuję!