



POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Piotr Anuśkiewicz

Nr albumu 24067

Kolejki wiadomości w systemach architektury mikroserwisów

Praca magisterska

dr inż. Mariusz Trzaska

Warszawa, luty 2023

Streszczenie

Praca dotyczy kolejek wiadomości i ich zastosowań. Obejmuje Apache Kafka, RabbitMQ, Apache RocketMQ, Apache ActiveMQ, NSQ, Amazon SQS. Opisuje zagadnienia związane z kolejkami, charakterystyczne cechy każdego z wybranych systemów, ich instalację z wykorzystaniem technologii Docker oraz implementację aplikacji klienckich w języku Java, z wykorzystaniem oficjalnych dedykowanych bibliotek. Do monitorowania pracy brokerów, producentów i konsumentów wykorzystany został Prometheus. Dokonano przeglądu dedykowanych eksporterów metryk i zaimplementowano własne. Dysponując systemem monitoringu, przeprowadzono doświadczenia z udziałem brokerów i aplikacji klienckich, badając poszczególne parametry konfiguracyjne. Dokonano subiektywnego podsumowania, wyciągając wnioski z doświadczeń z każdym z systemów kolejkowych – zostały uwzględnione m. in. złożoność konfiguracji, dodatkowe funkcje czy integracje z popularnymi narzędziami.

Słowa kluczowe: mikroserwisy, kolejki wiadomości, point-to-point, publish/subscribe, Docker, Prometheus, Java

Spis treści

1. WSTĘP.....	5
1.1. Cel pracy	5
1.2. Wybrane rozwiązania.....	5
1.3. Organizacja pracy	5
2. SYSTEMY KOLEJKOWE W ARCHITEKTURZE MIKROSERWISÓW	7
2.1. Monolit a mikroserwisy	7
2.2. Kolejki wiadomości	8
2.3. Wzorzec point-to-point	8
2.4. Wzorzec Publish/Subscribe.....	9
2.5. Model hybrydowy	10
2.6. Przewaga kolejek wiadomości nad HTTP	10
3. PRZEGLĄD SYSTEMÓW KOLEJKOWYCH	12
3.1. Kafka.....	12
3.2. RabbitMQ	13
3.3. RocketMQ.....	13
3.4. ActiveMQ	14
3.5. NSQ	15
3.6. SQS	16
3.7. Podsumowanie	17
4. PRZEGLĄD ISTNIEJĄCYCH PORÓWNAŃ.....	18
4.1. G2 - Best Message Queue Software [46]	18
4.2. StackShare – What are the best Message Queue Tools? [48]	19
4.3. PeerSpot – Best Message Queue (MQ) Software [47].....	19
4.4. Liderzy istniejących porównań	20
5. INSTALACJA I KONFIGURACJA BROKERÓW WIADOMOŚCI.....	22
5.1. Kafka.....	22
5.2. RabbitMQ	23
5.3. RocketMQ.....	25
5.4. ActiveMQ	27
5.5. NSQ	28
5.6. SQS	29
6. IMPLEMENTACJA APLIKACJI KLIENCKICH	31
6.1. Kafka.....	32
6.1.1. Producent.....	32
6.1.2. Konsument	34
6.1.3. Integracja z Apache Camel.....	35
6.1.4. Kafka Streams.....	35
6.2. RabbitMQ	37
6.2.1. Producent point-to-point	37
6.2.2. Producent Pub/Sub	39
6.2.3. Konsument point-to-point	41
6.2.4. Konsument Pub/Sub.....	41
6.3. RocketMQ.....	42
6.3.1. Producent.....	42
6.3.2. Konsument	43
6.4. ActiveMQ	44
6.4.1. Producent.....	44

6.4.2.	<i>Konsument</i>	45
6.5.	NSQ	46
6.5.1.	<i>Producent</i>	46
6.5.2.	<i>Konsument</i>	47
6.6.	SQS	48
6.6.1.	<i>Producent</i>	49
6.6.2.	<i>Konsument</i>	50
7.	INSTALACJA I KONFIGURACJA NARZĘDZI DO MONITOROWANIA.....	51
7.1.	Prometheus.....	51
7.2.	Grafana.....	52
7.3.	Eksportery metryk.....	54
7.3.1.	<i>Kafka</i>	54
7.3.2.	<i>RabbitMQ</i>	55
7.3.3.	<i>RocketMQ</i>	55
7.3.4.	<i>ActiveMQ</i>	56
7.3.5.	<i>NSQ</i>	57
7.3.6.	<i>SQS</i>	59
7.3.7.	<i>Aplikacje klienckie – Producer i Consumer</i>	59
8.	PORÓWNANIE SYSTEMÓW KOLEJKOWYCH.....	61
8.1.	Dostępność dokumentacji	61
8.2.	Obraz Docker	61
8.3.	Instalacja i niezbędna konfiguracja	61
8.4.	Integracja z narzędziami do monitorowania	62
8.5.	Operacje i debugowanie	62
8.6.	Rozmiary obrazów Docker	62
9.	TESTY NA ŚRODOWISKU	64
9.1.	Kafka – przepustowość producenta w zależności od ACKS.....	64
9.2.	Kafka – przepustowość producenta w zależności od rozmiaru wiadomości.....	64
9.3.	Kafka – łączna przepustowość produkcji w zależności od liczby producentów	64
9.4.	Kafka – przepustowość produkcji w zależności od liczby partycji <i>topica</i>	65
9.5.	Kafka – przepustowość konsumpcji klientów w jednej i kilku grupach	65
9.6.	RabbitMQ – przepustowość producenta w zależności od rodzaju kolejki – <i>durability</i>	65
9.7.	RabbitMQ – przepustowość produkcji <i>point-to-point</i> a <i>Pub/Sub</i>	65
9.8.	RocketMQ – przepustowość producenta w zależności od rozmiaru wiadomości.....	65
9.9.	RocketMQ – łączna przepustowość produkcji w zależności od liczby producentów	66
9.10.	ActiveMQ – przepustowość produkcji <i>point-to-point</i> a <i>Pub/Sub</i>	66
9.11.	NSQ – przepustowość produkcji buforowanej i niebuforowanej.....	66
9.12.	SQS – łączna przepustowość produkcji w zależności od liczby producentów	66
9.13.	SQS – łączna przepustowość konsumpcji w zależności od liczby konsumentów.....	67
9.14.	Porównanie wyników.....	67
10.	PODSUMOWANIE.....	81
	LITERATURA.....	82

1. Wstęp

Termin „mikroserwisy” zyskał ostatnimi laty na popularności, wraz ze wzrostem świadomości na temat znaczenia architektury systemu dla jego utrzymania i rozwoju. Niniejsza praca traktuje na temat istotnych usług, spinających ze sobą mikroserwisy, czyli kolejek wiadomości.

Kolejki nie są niczym nowym ani na pierwszy rzut oka specjalnie skomplikowanym. Okazuje się jednak, że to pozornie proste zagadnienie skrywa wiele niuansów, o których warto wiedzieć dobierając brokera wiadomości do swojego systemu informatycznego. Liczne artykuły i publikacje korzystają z różnych definicji pojęć związanych z kolejkami, co wywołuje u czytelnika jeszcze większą dezorientację.

1.1. Cel pracy

Celem pracy jest przybliżenie czytelnikowi najważniejszych zagadnień związanych z kolejkami wiadomości i porównanie popularnych obecnie rozwiązań.

Porównanie przeprowadzono na podstawie doświadczeń nabytych podczas pracy z kolejkami. Wzięto pod uwagę nakład pracy potrzebnej do instalacji i konfiguracji brokerów, a także do napisania aplikacji klienckich. Uwzględniono obszerność dokumentacji dostępnej w Internecie, dodatkowe funkcjonalności i wygodę użytkowania.

1.2. Wybrane rozwiązania

W pracy opisano i porównano ze sobą sześć brokerów wiadomości:

- Kafka,
- RabbitMQ,
- RocketMQ,
- ActiveMQ,
- NSQ,
- Amazon SQS.

Pięć spośród nich to darmowe rozwiązania open-source. Amazon SQS jest jedyną płatną i chmurową kolejką wiadomości. Jego obecność w porównaniu ma służyć weryfikacji czy rozwiązania chmurowe są realną alternatywą dla samodzielnie wdrażanych kolejek.

Dla każdego z systemów kolejkowych napisane zostały proste aplikacje klienckie – producent i konsument, oraz eksportery metryk. Wykorzystano język Java wraz ze szkieletem Spring Boot. Niewielką część kodu napisano w języku Kotlin.

Do monitorowania pracy systemów kolejkowych wykorzystano stos Prometheus-Grafana.

Brokery wiadomości, aplikacje klienckie oraz narzędzia do monitoringu zainstalowano w postaci kontenerów – użyto technologii Docker.

1.3. Organizacja pracy

W rozdziale 2 wyjaśnione zostały pojęcia architektury mikroserwisów i kolejki wiadomości. Opisano podstawowe wzorce spotykane we współczesnych brokerach, przedstawiono zalety komunikacji między serwisowej za pośrednictwem kolejek.

W kolejnym rozdziale opisano wybrane systemy kolejkowe, przyporządkowano odpowiadające im wzorce i wyszczególniono cechy charakterystyczne. W rozdziale 4 dokonano przeglądu istniejących porównań kolejek wiadomości, dostępnych w Internecie.

W rozdziałach 5 i 6 opisano instalację i konfigurację brokerów z wykorzystaniem technologii Docker oraz implementację prostych aplikacji producenta i konsumenta. W rozdziale 7 scharakteryzowano wykorzystywany stos Prometheus-Grafana i przedstawiono użyte eksportery metryk.

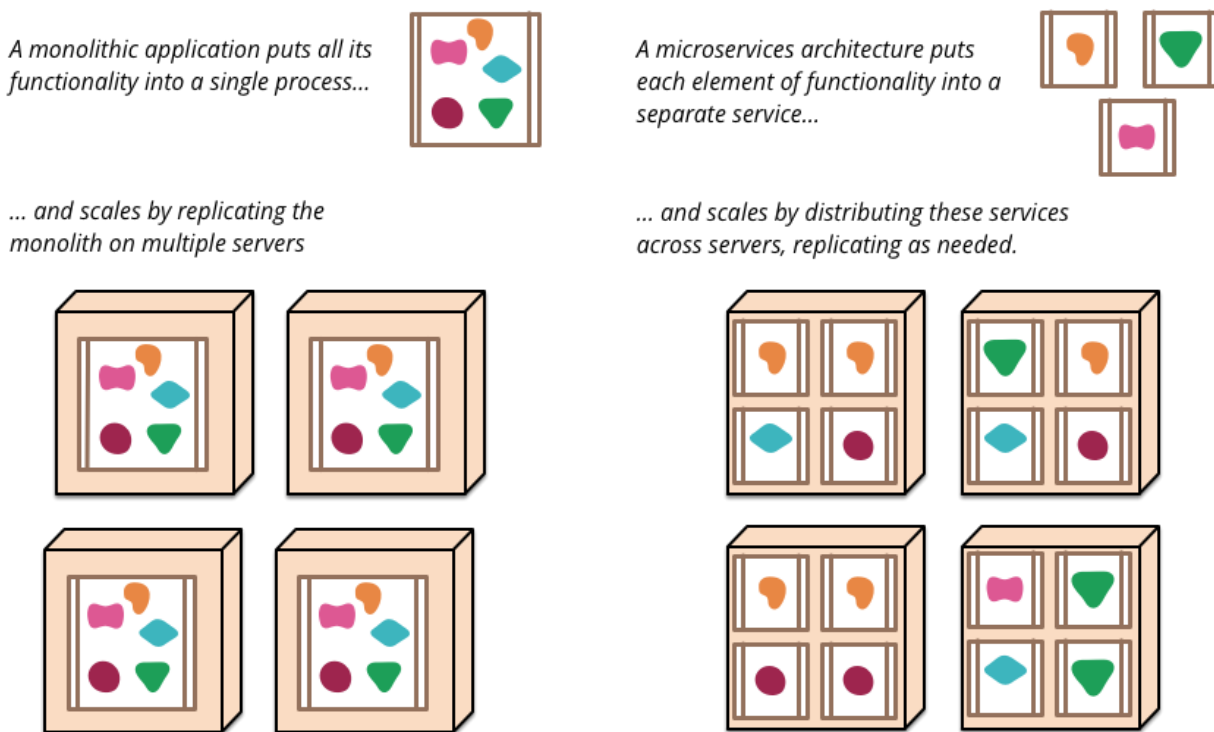
Rozdział 8 zawiera porównania wydajności dla różnych systemów kolejkowych i ich konfiguracji. W ostatnim rozdziale podsumowano doświadczenia i wyciągnięto wnioski na temat każdego z brokerów wiadomości.

2. Systemy kolejkowe w architekturze mikroserwisów

W niniejszym rozdziale wyjaśniono czym są architektury monolitu i mikroserwisów. Wy tłumaczono znaczenie odpowiedniej komunikacji między serwisami – synchronicznej lub asynchronicznej. Zdefiniowano kolejkę wiadomości i opisano jej podstawowe wzorce. Przedstawiono zalety kolejek w porównaniu do API HTTP.

2.1. Monolit a mikroserwisy

Monolit to złożona aplikacja, realizującą wszystkie swoje funkcjonalności w obrębie jednego procesu (programu, kontenera). Takie podejście do budowania systemu informatycznego wydaje się naturalnym, ale niesie ze sobą kilka wad.



Rysunek 1 - porównanie skalowania horyzontalnego aplikacji monolitu i mikroserwisów. Źródło: [34].

Aplikacje rzadko pracują pod stałym obciążeniem, jest ono raczej zmienne w czasie. Infrastruktura (serwery, wirtualne maszyny w chmurze), na której oprogramowanie jest uruchomione, kosztuje pieniądze, więc konieczne jest skalowanie. Wertykalne skalowanie aplikacji ma swój limit, po osiągnięciu którego pozostaje skalowanie horyzontalne – uruchamianie dodatkowych jej instancji. Problem w tym, że poszczególne moduły (interfejsy, przepływy) mogą być nierównomiernie obciążone, więc replikowanie ciężkiego monolitu jest mało wydajne (patrz: Rysunek 1). Rozwiązaniem tego problemu jest podzielenie aplikacji na mniejsze, niezależne od siebie wzajemnie jednostki – mikroserwisy [34].

Wyżej wspomniana niezależność jest inną istotną zaletą architektury mikroserwisów – nieoczekiwany błąd w przetwarzaniu, powstały w wyniku pomyłki programisty albo nieprzewidzianego scenariusza, nie spowoduje awarii całego systemu, a jedynie wyłączy kilka jego funkcjonalności z użytku, co ograniczy straty. Inną zaletą jest większa łatwość utrzymania takiej aplikacji – kod poszczególnych komponentów jest prostszy, a co za tym idzie – łatwiejszy do zrozumienia [2].

Architektura mikroserwisów ma jednak swoje wady. Należą do nich m. in.:

- więcej niezależnych aplikacji to bardziej skomplikowana procedura budowania kodu i wdrażania na środowisko,
- wiele aplikacji pracujących na tym samym źródle danych może oznaczać problem ze spójnością tych danych,
- skoro komponenty są niezależne i mogą pracować nawet na różnych serwerach, pojawia się problem komunikacji pomiędzy nimi (z którym nie mamy do czynienia w przypadku architektury monolitu).

Skoro mowa o komunikacji pomiędzy komponentami – według [34], mikroserwisy zazwyczaj komunikują się ze sobą za pośrednictwem protokołu HTTP albo kolejek wiadomości. Debata o wyższości jednego sposobu nad drugim jest w rzeczywistości debatą między komunikacją synchroniczną (HTTP) a asynchroniczną (kolejki wiadomości). Wyłonienie zwycięzcy jest niemożliwe, ponieważ oba sposoby znajdują swoje zastosowanie w różnych przypadkach użycia [5].

2.2. Kolejki wiadomości

Kolejka wiadomości (ang. *Message Queue*, *MQ*) to forma asynchronicznej komunikacji między programami. Wiadomości wysłane na kolejkę zostają przez nią przechowane do czasu odczytania i/lub usunięcia (a więc kolejka wiadomości musi być trwała¹) [11][39][40].

Zastosowanie kolejek wiadomości najczęściej wymaga wykorzystania serwera (w dalszej części pracy nazywanego brokerem), który pośredniczy w komunikacji i przechowuje odebrane komunikaty.

2.3. Wzorzec point-to-point

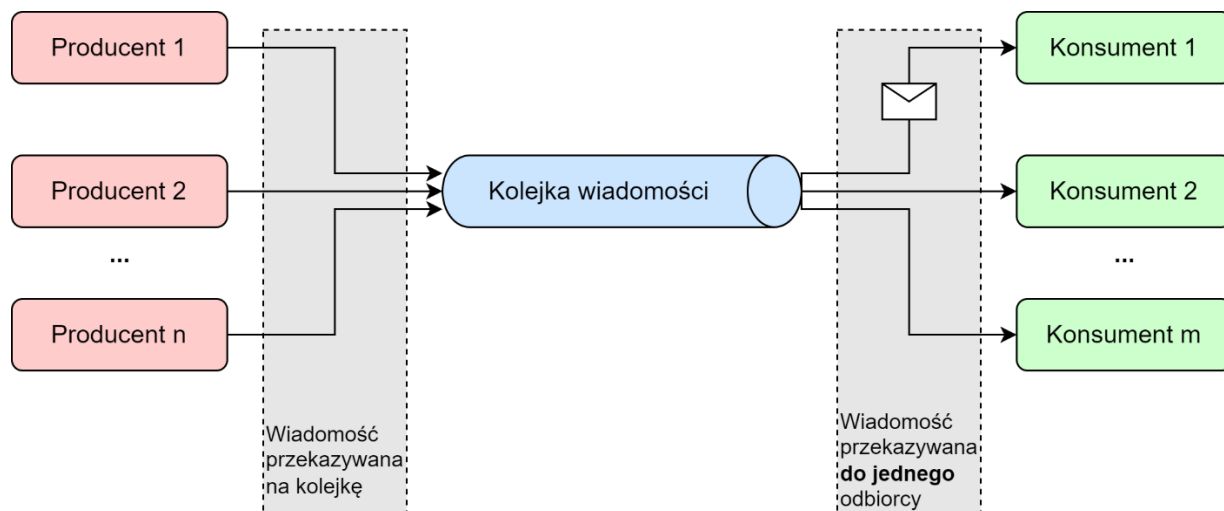
Modelem *point-to-point* nazywa się prostą kolejkę FIFO, która przekazuje każdą wiadomość tylko do jednego z połączonych odbiorców. Ten wzorzec stosuje się w sytuacjach, w których każda wiadomość powinna zostać przetworzona jednokrotnie. [39][40] Wzorzec *point-to-point* przedstawiono na Rysunku 2.

Wzorzec nie wyklucza równoczesnego czytania wiadomości z kolejki przez kilku klientów – w takim przypadku dzielą się pracą, co ma zastosowanie w skalowaniu horyzontalnym.

¹ Trwałość jest istotną cechą charakteryzującą kolejki wiadomości – a właściwie cechami, ponieważ tłumaczy się na dwa terminy w języku angielskim mające w kontekście kolejek wiadomości zgoła odmienne znaczenie.

Durability (o które chodzi w tym przypadku) oznacza, że wiadomość jest przechowywana podczas nieobecności konsumenta, aby mógł ją odczytać po powrocie.

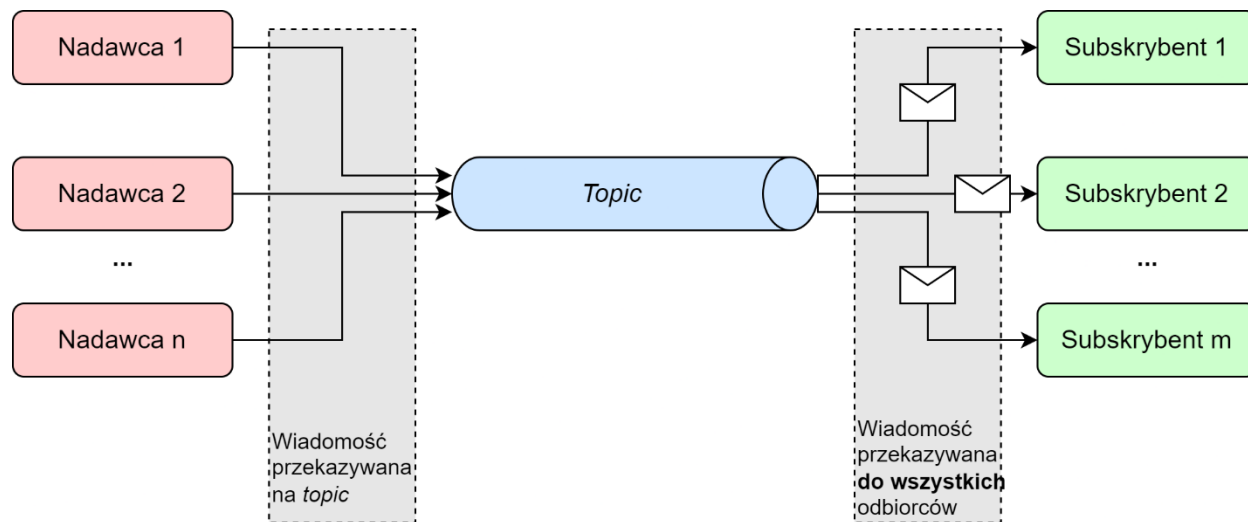
Persistence określa, czy otrzymana wiadomość jest przez kolejkę wiadomości zapisywana przed przekazaniem do konsumenta. Źródło: [39], s. 3-4.



Rysunek 2 - graficzne przedstawienie modelu point-to-point

2.4. Wzorzec Publish/Subscribe

Wzorzec *Publish/Subscribe* (zwany również *Pub/Sub*) rozdziela nadawcę i odbiorcę wiadomości (we wzorcu nazywanych nadawcą, ang. *publisher*, i subskrybentem). Nadawca publikuje wiadomość na *topic* bez wiedzy o tym, kto tą wiadomość odczyta. Każdy subskrybent odczytuje wiadomość z *topica* (patrz: Rysunek 3) [10].



Rysunek 3 - graficzne przedstawienie modelu Publish/Subscribe

Wiadomości są jednak przekazywane do każdego subskrybenta *topica*, więc nie ma mowy o dzieleniu się między nimi pracą, a raczej o wykonywaniu dodatkowej.

2.5. Model hybrydowy

Wiele współczesnych systemów kolejkowych jest implementacją modelu hybrydowego, który łączy cechy *point-to-point* i *Pub/Sub*. W najczęściej spotykanej implementacji (np. Kafka, RocketMQ, NSQ) wiadomości wysyłane są do jednej lub wielu grup, a w obrębie grupy są odczytywane przez jednego subskrybenta. Celnym określeniem takiego modelu jest *once-per-interested-party* (patrz: [39], s. 5). Rysunek 4 ilustruje działanie takiego modelu hybrydowego.

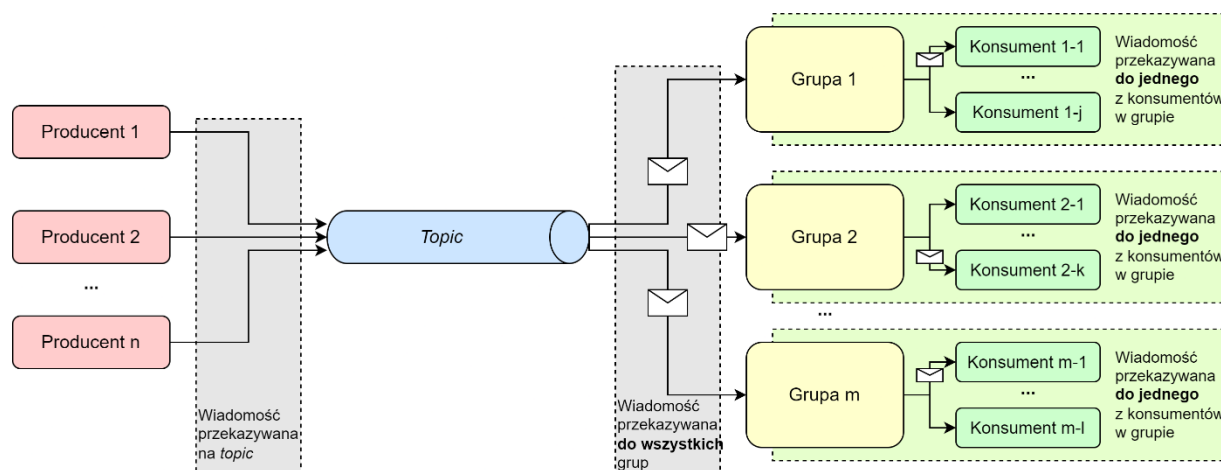
2.6. Przewaga kolejek wiadomości nad HTTP

Zaletą wykorzystania HTTP jest brak brokera, ale wiąże się z bezstanowością (o ile komponent nie ma np. wbudowanej lekkiej bazy danych). Jeżeli jeden z mikroservisów jest niedostępny, wszystkie inne, zależne od niego mikroservisów nie mogą działać. HTTP ma przewagę w przypadku integracji aplikacji z zewnętrznymi systemami – jest to uniwersalny protokół i nie wymaga dostosowania interfejsu do danej kolejki [43]. Wykorzystanie wiadomości nie wyklucza jednak zastosowania w systemie również komunikacji HTTP.

Komunikacja poprzez wiadomości wiąże się z ustanowieniem tzw. brokera – serwera obsługującego wiadomości i produkujących/konsumujących je klientów. Broker może zapewniać persystencję wiadomości – wówczas przerwa w pracy jednego komponentu nie zaburza działania całego systemu.

Stosowanie komunikacji HTTP między mikroservisami może doprowadzić do powstania długich łańcuchów zapytań (ang. *request chains*) [43]. Na przykład: komponent A wysłał zapytanie do komponentu B i oczekuje na odpowiedź. Komponent B procesuje to zapytanie, ale wiąże się to z wysłaniem zapytania do komponentów C i D. Pojedyncza transakcja zakończy się dopiero kiedy komponent A otrzyma odpowiedź i zostanie przerwana w razie malfunkcji któregoś z serwisów.

Stosując kolejki wiadomości, powyższą transakcję można podzielić na wiele mniejszych, realizowanych przez jeden komponent, zaczynających się od odczytania wiadomości i kończących zapisaniem jej do tej samej lub innej kolejki.



Rysunek 4 - graficzne przedstawienie modelu hybrydowego

Model hybrydowy kolejek wiadomości daje dużą dowolność w skalowaniu horyzontalnym i dzieleniu *workloadu* w celu optymalizacji pracy systemu. Skalowanie horyzontalne poszczególnych

mikroserwisów aplikacji stanowi istotną przewagę takiej architektury nad architekturą monolitu [34]. Ponadto, serwisy nie muszą o sobie wzajemnie wiedzieć, wystarczy komunikacja z brokerem.

Wyżej wspomniana trwałość wiadomości ułatwia też znajdowanie problemów w działaniu systemu, takich jak błędy przetwarzania albo niska wydajność. Takie problemy są sygnalizowane przez nagromadzenie wiadomości na kolejce wejściowej komponentu (tzw. *lag*) [43].

3. Przegląd systemów kolejkowych

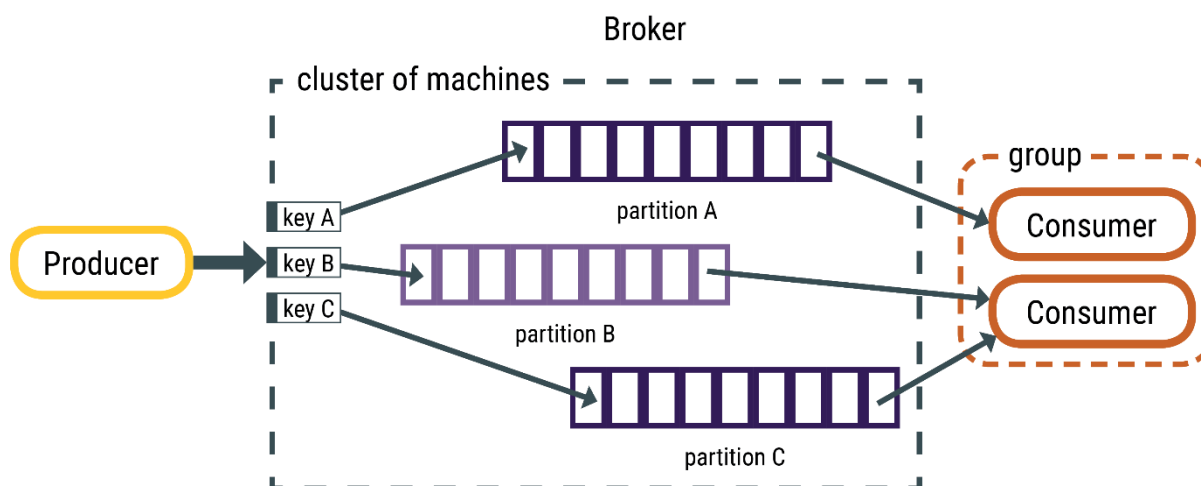
Poniższy rozdział zawiera krótki opis poszczególnych kolejek wiadomości i charakterystycznych dla nich cech. Kolejkom przyporządkowano wzorzec (*point-to-point*, *Pub/Sub*, hybrydowy).

3.1. Kafka

Kafka, pierwotnie wytworzona przez LinkedIn na własne potrzeby, jest obecnie rozwijana na licencji Apache przez Apache Software Foundation [39]. Charakteryzuje się skalowalnością, wysoką dostępnością (ang. *High Availability*) i wysoką przepustowością (ang. *high throughput*).

Kafka działa w modelu rozproszonym. Wspiera replikację danych i wysoką dostępność (ang. *High Availability*). Pomaga w tym Apache Zookeeper instalowany wraz z brokerem, zarządzający topologią i przechowujący najważniejsze metadane [24].

Kafka jest przykładem kolejki w modelu hybrydowym. Opiera się na *topicach*, na które wiadomości są wysyłane i grupach agregujących konsumentów (ang. *consumergroup*). Wiadomość jest odczytana po jednym razie przez każdą grupę (trafia do jednego z jej konsumentów). Kafka przechowuje oddzielny postęp w odczytywaniu wiadomości (nazywany *offsetem*) dla każdej grupy.



Rysunek 5 - model domenowy Kafki. Źródło: [19]

Kafka jest persystentna – wiadomości są zapisywane na dysk przed wysłaniem potwierdzenia odbioru² albo przekazaniem do konsumentów. Skonsumowane wiadomości nie są od razu usuwane i mogą zostać odczytane dowolną liczbę razy (poprzez manipulację *offsetami*). Administrator może skonfigurować retencję danych lub kompaktację – scalanie wiadomości według klucza.

² Jest tak przynajmniej w przypadku domyślnej konfiguracji producenta, ale może on też wybrać od ilu brokerów chce otrzymać potwierdzenie zanim uzna, że wiadomość trafiła bezpiecznie do celu – parametr *acks* (domyślnie 1). Mechanizm można wyłączyć, ustawiając *acks* = 0, ale też zwiększyć jego wartość, aby czekać na potwierdzenie od brokerów replikujących *topic*.

Każdy *topic* składa się z dowolnej liczby partycji, a każda partycja jest przechowywana w oddzielnej lokacji na dysku. Do każdej partycji może być równocześnie podłączony tylko jeden konsument – a więc liczba partycji stanowi maksymalną liczbę równoległych konsumentów. Konsumentem jest zazwyczaj jeden wątek. Podczas produkowania wiadomości na *topic*, wiadomości te są domyślnie dystrybuowane na różne partycje przy użyciu funkcji haszującej z klucza (aby zapewnić możliwie równomierną dystrybucję i przepływ wiadomości o takim samym kluczu w obrębie jednej partycji), choć producent może sam zdecydować na którą partycję wiadomość ma trafić. Model domenowy Kafki przedstawiony został na Rysunku 5.

W związku z powyższym, Kafka nie zapewnia odczytu wiadomości z *topica* w takiej samej kolejności, w jakiej zostały zapisane. Wymaga to uwzględnienia w projektowaniu systemu – kolejność otrzymywania danych nie powinna wpływać na jego końcowy stan (różnice w obsłudze wiadomości FIFO i nieuporządkowanych zostały szerzej opisane w [21], w sekcji *Model Comparison*). Wyjątkie jest *topic* o liczbie partycji równej 1 – Kafka zapewnia kolejność FIFO w obrębie partycji.

Wiadomości Kafki składają się z klucza, wartości, *timestampu* (daty i godziny wysłania wiadomości) i opcjonalnych nagłówków (ang. *headers*).

3.2. RabbitMQ

RabbitMQ również jest projektem open-source [41]. Został napisany w roku 2007, w języku Erlang. Początkowo opierał się jedynie na AMQP, obecnie wspiera kilka różnych protokołów transmisji wiadomości³.

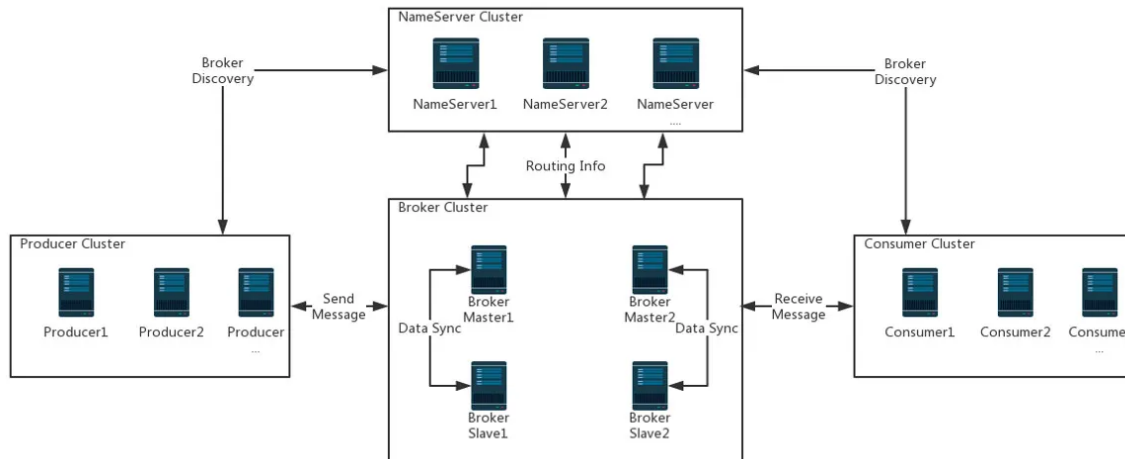
Podstawowa kolejka implementuje wzorzec *point-to-point*, ale wspierany jest też *Publish/Subscribe*. W przypadku *Pub/Sub* producent deklaruje wymianę (ang. *exchange*), na którą wysyłane są wiadomości, oraz klucz routingu. Klucz routingu służy do wysłania wiadomości tylko na niektóre kolejki w ramach tej samej wymiany. Sama komunikacja, mimo, że jest to model *Pub/Sub*, pod spodem opiera się na kolejkach *point-to-point*. Konsument i tak deklaruje zwykłą kolejkę, z której chce konsumować, ale może powiązać ją z odpowiednią wymianą [37].

Plusem RabbitMQ jest duża liczba dostępnych wtyczek, pozwalających w prosty sposób zainstalować serwer na przykład wraz z serwerem HTTP wystawiającym Management UI. W ten sposób broker nie jest obciążony niechcianymi dodatkowymi funkcjonalnościami i sam w sobie pozostaje lekki.

3.3. RocketMQ

RocketMQ zostało napisane przez Alibaba, wiodącą chińską firmę e-commerce, na własny użytek [36]. Od 2012 technologia jest udostępniona jako open-source. RocketMQ, podobnie jak Kafka i ActiveMQ jest przykładem hybrydowej kolejki wiadomości (mającej cechy *point-to-point* i *Pub/Sub*) [33].

³ Pełna lista wspieranych protokołów jest dostępna w oficjalnej dokumentacji: [28]



Rysunek 6 - diagram topologii klastra RocketMQ. Źródło: [36]

RocketMQ, podobnie jak Kafka, domyślnie wspiera model rozproszony. Klaster RocketMQ składa się z *nameserverów* i brokerów. Producent albo konsument, chcąc podłączyć się do brokera, wysyła najpierw zapytanie do *nameservera*, a w odpowiedzi otrzymuje adres brokera, który przechowuje żądane dane (patrz: Rysunek 6).

Organizacja RocketMQ w dużym stopniu przypomina organizację Kafki. *Topic* dzieli się na kolejki wiadomości, a nie partycje jak w przypadku Kafki, ale konsumenci łączący się do *topiców* również są pogrupowani w grupy (ang. *consumergroup*). Rysunek 7 przedstawia model domeny RocketMQ.

Producent RocketMQ może wysłać wiadomość na *topic* na jeden z trzech sposobów:

- synchronicznie,
- asynchronicznie,
- *one-way*.

Innymi interesującymi wariacjami są:

- wysyłanie wiadomości FIFO (ang. *ordered sending*),
- wysyłanie wiadomości z opóźnieniem (ang. *delayed sending*),
- wysyłanie wiadomości w paczkach (ang. *batch sending*),
- wysyłanie wiadomości w transakcjach.

Do każdej z powyższych funkcjonalności nietrudno dobrać jakiś przypadek użycia [32].

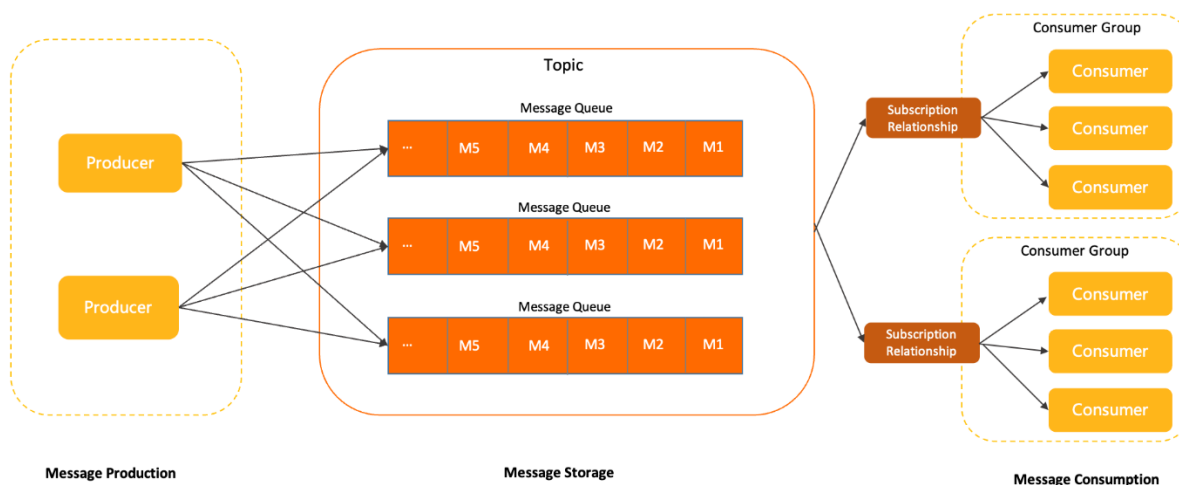
3.4. ActiveMQ

Najstarszy spośród opisywanych systemów kolejkowych, stworzony przez firmę LogicBlaze w roku 2004 jako pierwsze popularne rozwiązanie open-source tego typu [39]. ActiveMQ ([12]) jest implementacją

JMS. Specyfikacja JMS nie narzuca protokołu komunikacji, na potrzeby tego systemu stworzono OpenWire⁴.

Choć ActiveMQ wspiera model rozproszony, to w przeciwieństwie do Kafki nie współpracuje z żadną usługą zarządzania topologią klastra (jak np. Zookeeper). Brokery mogą połączyć się w układ *Master-Slave* (zazwyczaj łączy się w ten sposób dwa brokery, patrz [39], s. 26), co zapewnia wysoką dostępność, ale ogranicza wzrost wydajności przez skalowanie horyzontalne, w porównaniu do konkurencji.

ActiveMQ, podobnie jak RabbitMQ, implementuje wzorce *point-to-point* i *Pub/Sub*, ale już nie model hybrydowy. Klienci mogą używać albo kolejek, albo *topiców* [15].



Rysunek 7 - model domenowy RocketMQ. Źródło: [29]

3.5. NSQ

NSQ ([25]) to skrót od „Natively Scalable Queue”. Kolejkę napisano w języku Go i jest rozwijana przez Bitly jako projekt open-source od 2013 roku. Założeniami rozwiązania miały być:

- lekkość (ang. *lightweight*),
- zapewnienie wysokiej dostępności i eliminacja pojedynczych punktów awarii,
- prostota użytkowania⁵.

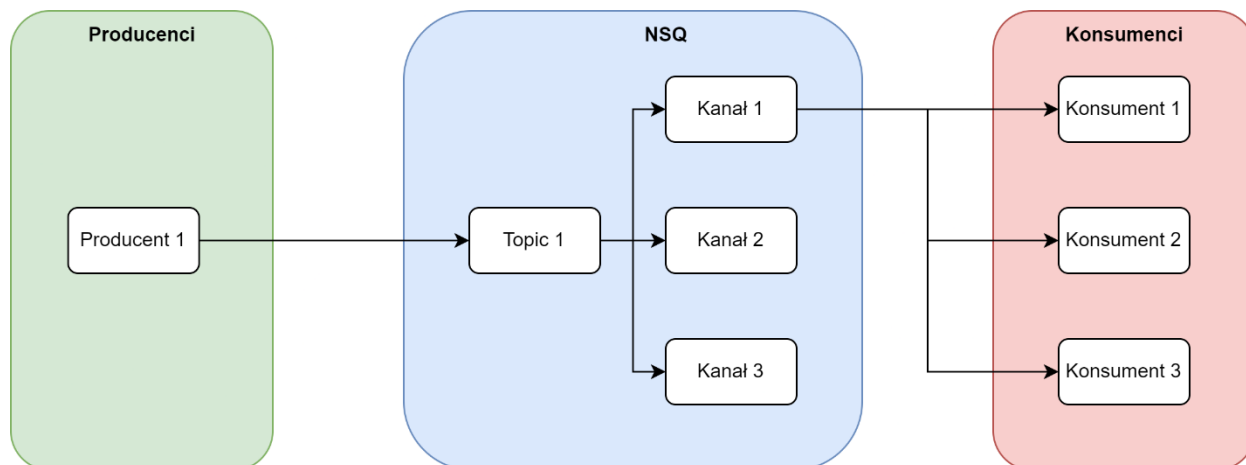
NSQ implementuje swój własny protokół komunikacyjny, opierający się na TCP.

Architektura NSQ przypomina nieco RocketMQ i Kafkę – wielowęzłowy klastr składa się z NSQLookupD (działającego analogicznie do Zookeepera i Nameservera) i brokera (NSQD). Dodatkowym, opcjonalnym komponentem jest NSQAdmin – serwer HTTP oferujący interfejs graficzny służący do zarządzania i monitorowania pracy klastra.

⁴ Obecnie ActiveMQ, podobnie jak RabbitMQ, wspiera wszystkie popularne protokoły: AMQP, STOMP, XMPP, MQTT. Źródło: [39].

⁵ W duchu poprzedniej kolejki wiadomości wykorzystywanej przez Bitly: Simplequeue, wchodzącej w pakiet narzędzi Simplehttp – <https://github.com/bitly/simplehttp/tree/master/simplequeue> (dostęp 2023-01-09)

Również podobna do Kafki i RocketMQ jest organizacja – NSQ jest hybrydową kolejką wiadomości. Rysunek 8 przedstawia model domenowy. Rolę analogiczną do grup konsumentów w konkurencyjnych rozwiązaniach pełnią kanały. Wiadomość wysłana na *topic* trafia do każdego kanału – jeżeli do kanału jest połączonych kilku konsumentów, tylko jeden z nich otrzymuje wiadomość.



Rysunek 8 - model domenowy NSQ. Źródło: własne, na podstawie dokumentacji NSQ – [25]

3.6. SQS

Jedyna w tym zestawieniu usługa chmurowa. Amazon SQS został udostępniony klientom w 2006 roku (2004 jako beta). Jest przykładem kolejki *point-to-point*.

Amazon oferuje dwa rodzaje kolejek wiadomości SQS [8]. Standardowa kolejka SQS nie zapewnia kolejności FIFO (jest to tzw. mechanizm *best-effort* – wiadomości zazwyczaj docierają w odpowiedniej kolejności, ale mogą się zdarzyć wyjątki). Nie zapewnia też dokładnie jednego dostarczenia wiadomości, zamiast tego oferując przynajmniej jedno dostarczenie. Zachowanie kolejności wiadomości i dokładnie jedno dostarczenie są oferowane przez kolejkę typu FIFO, która jest droższa w użytkowaniu. Kolejki FIFO mają za to ograniczoną przepustowość – według dokumentacji wspierane jest do 300 operacji na sekundę⁶.

Maksymalny wspierany rozmiar wiadomości to 256 kB, przy czym przy wycenie uwzględniane jest każde 64 kB wiadomości (a więc za dwie wiadomości po 64 kB każda płaci się tyle samo, co za jedną 128 kB-ową).

Podobnie jak w przypadku Kafki, wiadomości nie są automatycznie usuwane z kolejki od razu po odczytaniu. O ile nie skasuje ich konsument, z kolejki znikną dopiero po konfigurowalnym czasie retencji. Ten można ustawić pomiędzy 1 minutą a 14 dniami (domyślnie 4 dni).

Podczas standardowego cyklu pracy producenci i konsumenci kolejki SQS korzystają z trzech typów operacji (transakcji):

⁶ Operacją jest wysłanie wiadomości, odebranie wiadomości lub usunięcie wiadomości z kolejki – a więc 300 operacji odpowiada faktycznie 100 przesłanym wiadomościom. W dokumentacji nadmieniono, że w celu zwiększenia przepustowości można wiadomości pakować w paczki (maksymalnie 10, a więc przepustowość wzrośnie do 1000 wiadomości na sekundę) lub włączyć tryb wysokiej przepustowości (ang. *high throughput mode*), który podnosi przepustowość do 1000 wiadomości bez paczkowania (lub 10000 z paczkowaniem). Tryb wysokiej przepustowości jest dostępny w niektórych regionach AWS [8].

- wyślij wiadomość (ang. *Send Message*),
- odczytaj wiadomość (ang. *Receive Message*),
- usuń wiadomość (ang. *Delete Message*).

Oznacza to, że wiadomości można przysyłać bez konieczności stałego utrzymania połączenia. Cykl życia wiadomości jest następujący:

- 1) Producent wysyła wiadomość na kolejkę.
- 2) Konsument odczytuje wiadomość z kolejki.
- 3) Konsument może usunąć wiadomość z kolejki po odczytaniu – w ten sposób wiadomość nie zostanie ponownie skonsumowana.
- 4) Jeżeli wiadomości nie skasowano po przeczytaniu, zostanie ona usunięta automatycznie po upływie czasu retencji.

Co istotne, aby konsument mógł sprawdzić, czy jakaś wiadomość nie czeka w kolejce, musi skorzystać z operacji odczytania wiadomości. Korzysta więc z API SQS nawet wtedy, kiedy nic nie przetwarza. Rzeczywistą atomową jednostką, za którą płaci klient, jest właśnie pojedyncze skorzystanie z operacji. W praktyce oznacza to, że nawet bezczynność konsumenta wiąże się z ponoszeniem kosztów użytkowania kolejki. Amazon w swojej dokumentacji proponuje wykorzystanie mechanizmu rzadkiego odpytywania (*long polling*) – jeżeli w momencie odpytania kolejki nie oczekuje na niej żadna wiadomość, interwał przed kolejnym zapytaniem będzie znacznie dłuższy (oznacza to jednak, że następna wiadomość może zostać odczytana z dużym opóźnieniem).

3.7. Podsumowanie

W Tabeli 1 zestawiono cechy opisanych wyżej systemów kolejkowych.

Tabela 1 – porównanie wybranych systemów kolejkowych

Broker	Wzorzec	Usługa zarządzania topologią	Praca w modelu rozproszonym	Rok powstania	Bezpłatny
Kafka	hybrydowy	tak – Zookeeper	tak	2011	tak
RabbitMQ	<i>point-to-point, pub/sub</i>	nie	nie	2007	tak
RocketMQ	hybrydowy	tak – nameserver	tak	2012	tak
ActiveMQ	<i>point-to-point, pub/sub</i>	nie	tak	2004	tak
NSQ	hybrydowy	tak – nsqlookupd	tak	2013	tak
SQS	<i>point-to-point</i>	nie dotyczy	nie dotyczy	2006	nie

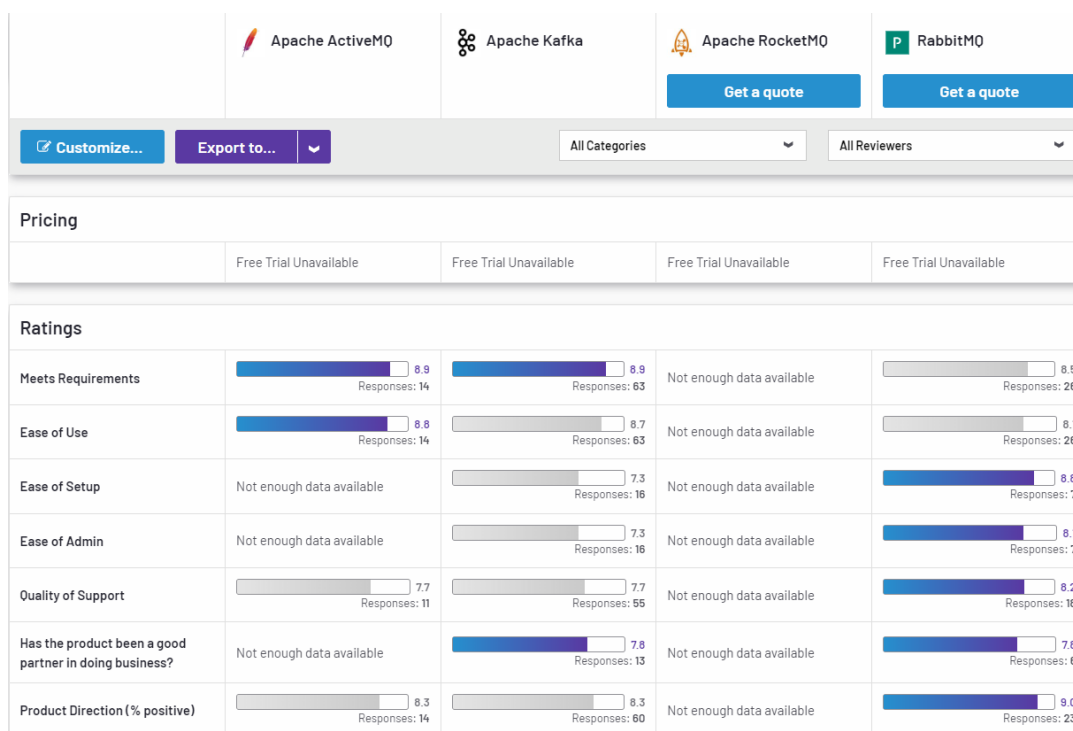
4. Przegląd istniejących porównań

Dostępnych jest wiele rozwiązań płatnych i open-source. Podjęcie decyzji o wyborze technologii mają ułatwić liczne porównania i rankingi dostępne w Internecie. Zaletami wyboru popularnego narzędzia są na przykład:

- duża dostępność materiałów, dokumentacji i poradników,
- mnogość gotowych integracji z innymi popularnymi narzędziami,
- większe prawdopodobieństwo, że to narzędzie jest aktywnie rozwijane i doczeka się kolejnych usprawnień i funkcjonalności.

4.1. G2 - Best Message Queue Software [46]

G2 jest portalem z recenzjami oprogramowania biznesowego. Opublikowane porównanie obejmuje 52 narzędzia, dla których zawiera opinie użytkowników (wraz z krótkimi opisami organizacji, dla których pracują, a które wykorzystują recenzowane narzędzia), cechy produktów, a także wizualizuje miejsce, które zajmuje dany system w dwuwymiarowej skali „Satysfakcja Użytkowników – Obecność na Rynku”.



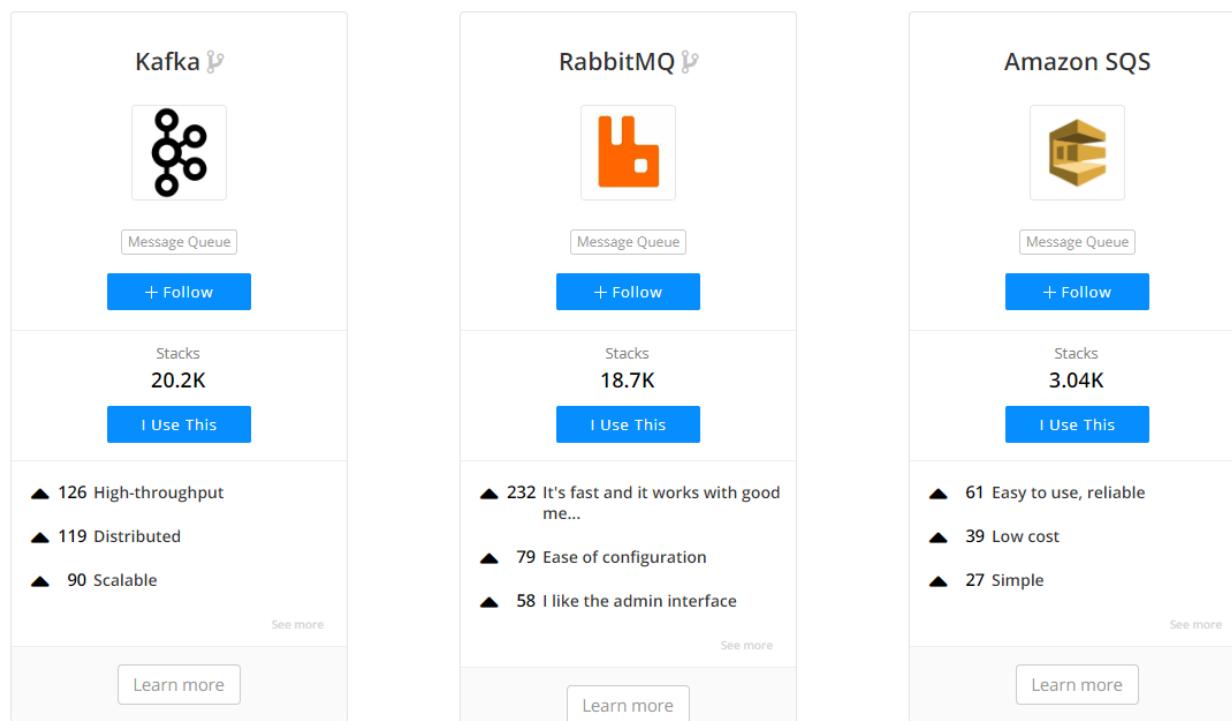
Rysunek 9 - tabela wygenerowana przez portal G2 dla czterech wybranych systemów kolejkowych. Źródło: [46]

Dla każdego narzędzia G2 oblicza też ogólną ocenę (nazywaną tutaj G2 Score) na podstawie recenzji użytkowników i m. in. danych zebranych z sieci społecznościowych. Lista zawiera kilka pozycji, jak na przykład MuleSoft Anyconnect Platform, Nasted czy Azure Scheduler, które nie są kolejkami wiadomości, a jedynie oferują zbliżone funkcjonalności lub integracje z popularnymi brokerami. Strona pozwala na

wygenerowanie zestawienia wybranych cech kilku rozwiązań (przykładowe zestawienie przedstawiono na Rysunku 9).

4.2. StackShare – What are the best Message Queue Tools? [48]

W przeciwieństwie do G2, w którego porównaniu nie stwierdzono jednoznacznie, który broker wiadomości jest najlepszy, Stackshare szereguje poszczególne systemy pod względem ocen użytkowników. Na podium znajdują się Apache Kafka, RabbitMQ oraz Amazon SQS. Każdy z systemów kolejkowych jest opisany serią etykiet-zalet, na które głosują użytkownicy portalu, poprawiając pozycję ocenianego systemu w rankingu (co widać na Rysunku 10). Wylistowane są cechy każdego z brokerów, można wygenerować zestawienie dwóch lub więcej z nich. Użytkownik może też kliknąć w zakładkę *Jobs*, w której znajdzie listę ofert pracy zawierającej daną technologię w wymaganiach (choć wiele spośród odnośników nie działa).



Rysunek 10 - trzy najpopularniejsze kolejki przedstawione w zestawieniu StackShare wraz z cechami najczęściej używanymi do ich opisania. Źródło: [48]

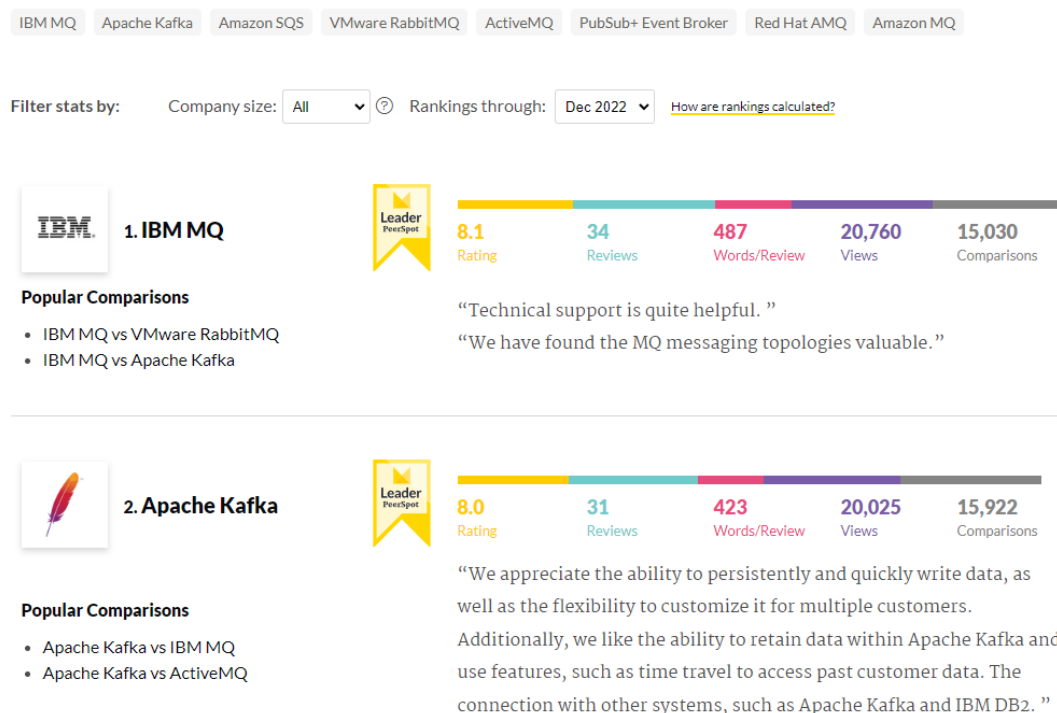
4.3. PeerSpot – Best Message Queue (MQ) Software [47]

Porównanie rozpoczyna się od krótkiego artykułu zawierającego definicję kolejki wiadomości i opisującego typowe zagadnienia z nią związane. Poniżej znajduje się ranking ośmiu najlepiej ocenianych systemów kolejkowych (pokazany na Rysunku 11). Wyniki można filtrować według wielkości organizacji, w której dany system zaimplementowano, a także według miesiąca/kwartału – dostępne są rankingi historyczne – użytkownik może prześledzić, jak zmieniała się popularność systemów kolejkowych na

przestrzeni lat. Udostępniono opis metody wyznaczania ogólnej oceny systemu, uwzględnia ona między innymi średnią ocen użytkowników i średnią liczbę słów na recenzję.

Strona pozwala na wygenerowanie porównania dwóch systemów kolejkowych, składającego się głównie z ich zalet i wad, opisanych przez specjalistów wdrażających lub pracujących z danym oprogramowaniem. Zawierają ich subiektywne opinie oraz zastosowanie, jakie kolejki znalazły w ich projektach.

Top 8 Message Queue (MQ) Software ?



Rysunek 11 - dwie najpopularniejsze kolejki wiadomości według zestawienia PeerSpot. Źródło: [47]

4.4. Liderzy istniejących porównań

Z wcześniej wymienionych zestawień wynika, że do najpopularniejszych rozwiązań należą IBM MQ, RabbitMQ i Kafka, z których ostatnia zajmuje miejsce na podium we wszystkich trzech rankingach. W Tabeli 2 zestawiono najpopularniejsze kolejki na każdym z portali.

Tabela 2 - zestawienie liderów rankingów w opisywanych zestawieniach

	G2 (G2 Score)	StackShare	PeerSpot
1.	IBM MQ	Apache Kafka	IBM MQ
2.	Apache Kafka	RabbitMQ	Apache Kafka
3.	Apache Qpid	Amazon SQS	Amazon SQS
4.	RabbitMQ	Celery	ActiveMQ
5.	Azure Queue Storage	ActiveMQ	RabbitMQ
6.	TIBCO Rendezvous	Azure Service Bus	PubSub+ EventBroker
7.	Google Cloud Pub/Sub	MQTT	Red Hat AMQ
8.	Red Hat AMQ	Apache NiFi	Amazon MQ

5. Instalacja i konfiguracja brokerów wiadomości

Każdy z brokerów (a także usług pomocniczych, jak Zookeeper czy Admin) został zainstalowany w kontenerze Docker. Dla ułatwienia posłużono się narzędziem Docker-Compose⁷. Wykorzystano minimalną konfigurację, polegając głównie na tej domyślnej (*out-of-the-box*). Wybrano oficjalne obrazy dockerowe, o ile istnieją.

5.1. Kafka

Dostępnych jest kilka nieznacznie różniących się od siebie obrazów dockerowych, udostępnianych między innymi przez Confluent Inc. albo Bitnami. Większość obrazów dockerowych zawiera Kafka CLI⁸. Kafka instalowana jest razem z Apache Zookeeper, który przechowuje informacje o brokerach w klastrze, a także metadane *topiców* i grup.

Listing 1 - zawartość pliku konfiguracyjnego Docker-Compose, służącego do utworzenia kontenerów Kafki i Zookeepera

```
---
version: '3'
services:
  zookeeper:
    image: bitnami/zookeeper:3.6.4 # January 2023
    container_name: kafka-zookeeper
    restart: always
    environment:
      ALLOW_ANONYMOUS_LOGIN: yes

  Kafka:
    image: bitnami/kafka:3.1.2 # January 2023
    container_name: kafka-broker
    ports:
      - "9094:9094"
    depends on:
      - zookeeper
    restart: always
    environment:
      ALLOW_PLAINTEXT_LISTENER: yes
      KAFKA_ZOOKEEPER_CONNECT: 'kafka-zookeeper:2181'
      KAFKA_LISTENERS: L1://:9094,L2://:29092
      KAFKA_ADVERTISED_LISTENERS: L1://localhost:9094,L2://kafka-broker:29092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: L1:PLAINTEXT,L2:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: L2
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1

networks:
  default:
    external:
      name: cluster-network-1
```

⁷ Compose jest narzędziem umożliwiającym definiowanie i operowanie kontenerami przy wykorzystaniu plików konfiguracyjnych YAML. Szerzej opisany w oficjalnej dokumentacji Dockera – [20].

⁸ Czyli lekki zestaw narzędzi umożliwiających produkowanie i konsumowanie wiadomości, a także zarządzanie serwerem Kafki. Konfiguracja i przykłady użycia zostały opisane w oficjalnej dokumentacji Kafki: <https://docs.confluent.io/platform/current/tutorials/examples/clients/docs/kafka-commands.html> (dostęp 2022-12-06)

Wybrane obrazy – bitnami/kafka w wersji 3.1.2 (styczeń 2023) i bitnami/zookeeper w wersji 3.6.4 (styczeń 2023) – nie należą do najlżejszych, zajmując kolejno 534 MB i 509 MB (nieskompresowane).

Definicje obu kontenerów przedstawiono w Listingu 1. Zarówno Kafkę i Zookeepera można konfigurować poprzez przekazywanie wartości parametrów w zmiennych środowiskowych⁹. Zookeeper do działania wymaga modyfikacji tylko jednej zmiennej: `ALLOW_ANONYMOUS_LOGIN`, wyłączającej autentykację.

Podstawowa konfiguracja Kafki jest nieco bardziej skomplikowana:

- `ALLOW_PLAINTEXT_LISTENER` pozwala zdefiniować interfejsy nasłuchujące (ang. *listeners*) bez autentykacji i szyfrowania¹⁰ (domyślnie wyłączone).
- `KAFKA_ZOOKEEPER_CONNECT` zawiera adres Zookeepera zarządzającego klastrem.
- W trzech kolejnych zmiennych środowiskowych zdefiniowano rozgłaszane interfejsy nasłuchujące i skonfigurowano oba jako `PLAINTEXT`.
- Jeżeli zdefiniowano więcej niż jeden interfejs, wymagane jest wyspecyfikowanie który spośród interfejsów ma być wykorzystywany w komunikacji wewnętrznej między brokerami – `KAFKA_INTER_BROKER_LISTENER_NAME`.
- Ponieważ uruchomiono klastrowanie zawierający jedną instancję Kafki, dla prawidłowego działania należy zmienić `KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR` na 1 (domyślnie 3) – w przeciwnym razie Kafka będzie usiłowała replikować *topic* i może nie działać poprawnie.

5.2. RabbitMQ

Dostępny jest oficjalny obraz dockerowy. RabbitMQ nie wykorzystuje dodatkowych usług zarządzania topologią, więc instalacja polega na uruchomieniu jednego lub kilku kontenerów brokera.

RabbitMQ wspiera wtyczki. Można je włączyć podczas uruchomienia serwera (za pomocą pliku konfiguracyjnego zawierającego listę nazw wtyczek) lub podczas jego pracy, przy użyciu narzędzi CLI.

Definicję kontenera RabbitMQ przedstawiono w Listingu 2. Użyto oficjalnego obrazu w wersji 3.9.17 (maj 2022). Obraz jest stosunkowo lekki – 224 MB.

⁹ Listę parametrów wraz z opisami można znaleźć na przykład w dokumentacji obrazów na docker.io: <https://hub.docker.com/r/bitnami/zookeeper> (dostęp 2023-01-11) i <https://hub.docker.com/r/bitnami/kafka> (dostęp 2023-01-11)

¹⁰ Kafka wspiera cztery protokoły zabezpieczające:

- `PLAINTEXT` – brak autentykacji, brak szyfrowania
- `SASL_PLAINTEXT` – autentykacja SASL, brak szyfrowania
- `SASL_SSL` – autentykacja SASL, szyfrowanie SSL
- `SSL` – autentykacja i szyfrowanie SSL

Listing 2 - zawartość pliku Docker-Compose - minimalna definicja RabbitMQ

```
---
version: '3'
services:
  rabbitmq:
    image: rabbitmq:3.9.17 # May 2022
    ports:
      - "9095:5672" # rabbitmq
```

RabbitMQ można uruchomić podając tylko obraz i mapowanie portu. Zdecydowano się jednak użyć dwóch wtyczek:

- *rabbitmq_management*, uruchamiającą serwer HTTP z Management UI na porcie 15672
- *rabbitmq_prometheus*, uruchamiającą eksporter metryk na porcie 15692.

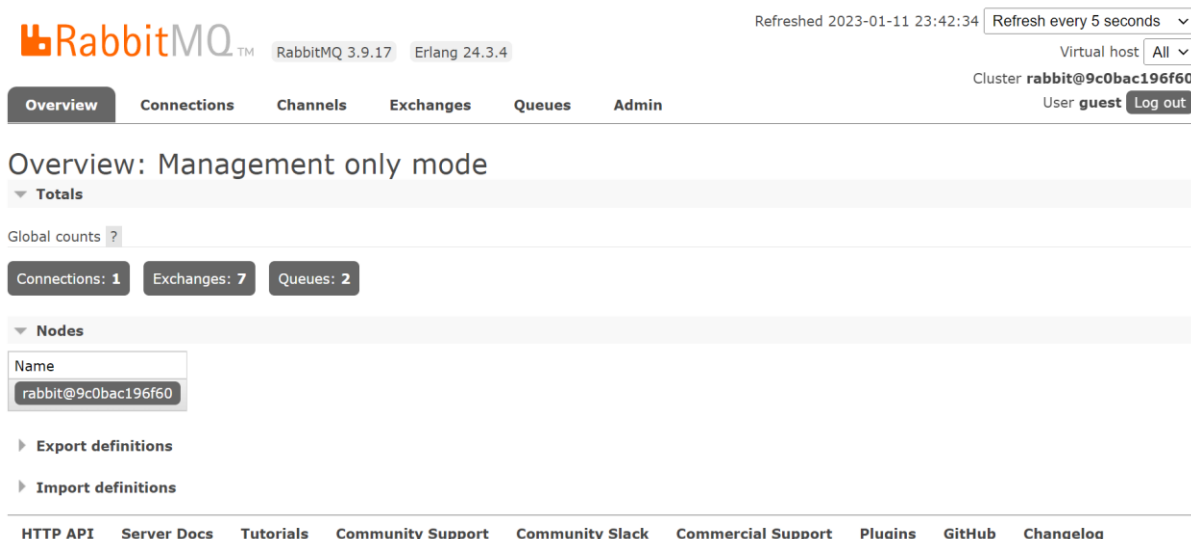
Do definicji kontenera dodano mapowanie wyżej wymienionych portów i wolumen (Listing 3), odzwierciedlający w kontenerze plik zawierający listę wtyczek.

Listing 3 - zawartość pliku Docker-Compose - definicja kontenera RabbitMQ

```
---
version: '3'
services:
  rabbitmq:
    image: rabbitmq:3.9.17 # May 2022
    container_name: rabbitmq
    ports:
      - "9095:5672" # rabbitmq
      - "9002:15692" # metrics
      - "9092:15672" # management
    volumes:
      - "/opt/masters-configs/rabbitmq/enabled_plugins:/etc/rabbitmq/enabled_plugins"
    restart: always

networks:
  default:
    external:
      name: cluster-network-1
```

Po uruchomieniu kontenera, oprócz kolejki działa również Management UI (Rysunek 12).



Rysunek 12 - ekran główny wbudowanego narzędzia administracyjnego RabbitMQ (Management UI)

5.3. RocketMQ

Istnieje oficjalny obraz dockerowy¹¹, ale jest też oficjalny projekt na GitHubie, pozwalający własnoręcznie wygenerować obraz, wybierając obraz bazowy (jedną z dwóch dystrybucji Linuxa – Alpine albo CentOS) i wersję kodu¹².

Skorzystano z generatora i wytworzono obraz w wersji 4.9.3 (marzec 2022), bazujący na Alpine. Jest dość lekki – zajmuje 175 MB. Broker RocketMQ instaluje się razem z Nameserverem, do obu kontenerów wykorzystuje się ten sam obraz z inną komendą wejściową.

Listing 4 zawiera definicję obu kontenerów. Nameserver domyślnie działa na porcie 9876, stąd ten port jest eksponowany. Broker eksponuje trzy porty do komunikacji z klientami, nameserverem i innymi brokerami. Aby producenci i konsumenci mogli połączyć się spoza sieci dockerowej, również one zostały zmapowane na porty zewnętrzne. Utworzono wolumeny – na logi dla kontenera Nameservera i na logi, dane i konfigurację Brokera.

¹¹ <https://hub.docker.com/r/apache/rocketmq> (dostęp 2023-01-04)

¹² <https://github.com/apache/rocketmq-docker> (dostęp 2023-01-04)

Listing 4 - zawartość pliku Docker-Compose - definicja kontenerów RocketMQ

```
---
version: '3'
services:
  nameserver:
    image: apacherocketmq/rocketmq:4.9.3-alpine
    container_name: rocketmq-nameserver
    ports:
      - "9096:9876"
    volumes:
      - "/opt/masters-configs/rocketmq/nameserver/logs:/home/rocketmq/logs"
    command: ./mqnamesrv

  broker:
    image: apacherocketmq/rocketmq:4.9.3-alpine
    container_name: rocketmq-broker
    ports:
      - "9097:9097"
      - "9098:9098"
      - "9099:9099"
    volumes:
      - "/opt/masters-configs/rocketmq/broker:/home/rocketmq "
    command: ./mqbroker -n rocketmq-nameserver:9876 -c ../conf/broker.conf
    depends_on:
      - nameserver

networks:
  default:
    external:
      name: cluster-network-1
```

Zawartość pliku konfiguracyjnego przedstawiono w Listingu 5. Poza konfiguracją domyślną zawiera nazwy klastra i brokera, flagę `autoCreateTopicEnable` pozwalającą na utworzenie nowego *topica* przez producenta wiadomości. Ustawiono dwa rozgłaszane adresy brokera (`localhost` i nazwę kontenera – dla sieci dockerowej), a także inny niż domyślny port do nasłuchiwania (9097).

Listing 5 - zawartość pliku konfiguracyjnego brokera RocketMQ

```
# Default:
brokerId = 0
deleteWhen = 04
fileReservedTime = 48
brokerRole = ASYNC_MASTER
flushDiskType = ASYNC_FLUSH
# Custom:
brokerClusterName = rocket-cluster-1
brokerName = rocket-broker-1
autoCreateTopicEnable = true
brokerIP1=localhost
brokerIP1=rocketmq-broker
listenPort=9097
```

5.4. ActiveMQ

W internecie nie jest dostępny oficjalny obraz dockerowy, więc stworzono go samodzielnie. Z oficjalnej strony projektu¹³ pobrano kod źródłowy w wersji 5.17.1 (kwiecień 2022). Utworzono definicję obrazu (plik Dockerfile przedstawiono na Listingu 6), bazującego na lekkim obrazie bazowym zawierającym środowisko uruchomieniowe Java (ActiveMQ 5.17.1 wymaga JRE w wersji 1.8 lub wyższej), do którego przekopiowano cały projekt.

Listing 6 - zawartość pliku Dockerfile, służącego do utworzenia obrazu ActiveMQ

```
FROM openjdk:11.0-jre-slim
COPY apache-activemq-5.17.1 /opt/apache-activemq-5.17.1
WORKDIR /opt/apache-activemq-5.17.1/bin
ENTRYPOINT ["./activemq", "console"]
```

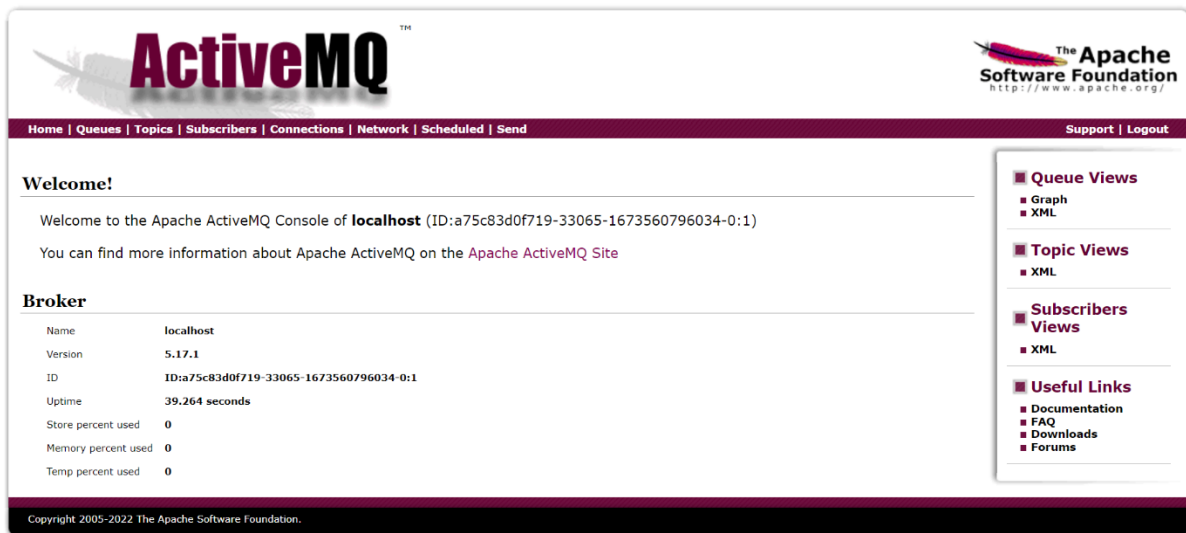
Po zbudowaniu obrazu przygotowano definicję kontenera w pliku Docker-Compose (Listing 7). Pliki konfiguracyjne znajdują się w katalogu „conf”, na którym utworzono wolumen, ale nie modyfikowano ustawień domyślnych, gdyż te okazały się być wystarczające do podstawowego używania kolejki.

Listing 7 - zawartość pliku Docker-Compose - definicja kontenera ActiveMQ

```
---
version: '3'
services:
  activemq:
    image: activemq:5.17.1
    container_name: activemq
    ports:
      - "9100:8161" # console
      - "9101:61616" # data - OpenWire protocol
    volumes:
      - "/opt/masters-configs/activemq/conf:/opt/apache-activemq-5.17.1/conf"
networks:
  default:
    external:
      name: cluster-network-1
```

Każdy spośród obsługiwanych protokołów komunikacyjnych wykorzystuje inny port TCP. Zmapowano 61616, na którym domyślnie działa OpenWire. Na 8161 jest wystawiany serwer HTTP z konsolą administracyjną (patrz: Rysunek 13). Pozwala modyfikować konfigurację *topiców* i kolejek, a nawet produkować na nie wiadomości. Oferuje też kilka prostych wykresów.

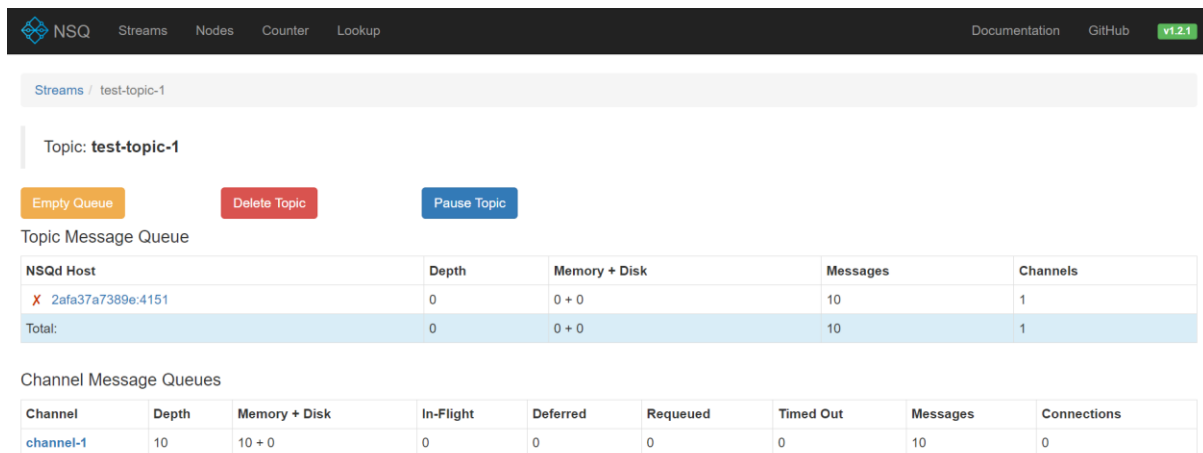
¹³ <https://activemq.apache.org/components/classic/download> (dostęp 2023-01-12)



Rysunek 13 - ekran główny konsoli administracyjnej ActiveMQ

5.5. NSQ

Jest dostępny oficjalny obraz dockerowy. Co ciekawe, ten sam obraz ma trzy zastosowania – w zależności od komendy startowej kontenera (ang. *entrypoint*), może zostać wykorzystany jako usługa zarządzająca topologią (*nsqllookupd*), broker (*nsqd*) lub narzędzie do administrowania (*nsqadmin*).



Rysunek 14 - konsola nsqadmin - ekran podsumowania przykładowego topica „test-topic-1”. Informuje nas, że na topic wysłano 10 wiadomości i jest on zasubskrybowany przez 1 kanał

Listing 8 zawiera definicje wszystkich trzech kontenerów, wykorzystujące oficjalny obraz w wersji v1.2.1 (sierpień 2021, od tamtego czasu nie wydano nowej wersji). Jediną konfiguracją, jaka jest wymagana do poprawnego działania są parametry w komendach uruchomieniowych brokera i admina, wskazujące na usługę nsqlookupd. Oprócz tego ustawienia domyślne są wystarczające.

Zarówno nsqlookupd jak i nsqd otwierają dwa porty TCP – „czysto” TCP 4150 i 4160 do komunikacji z aplikacjami klienckimi i 4151 oraz 4161, na którym działa HTTP API. Nsqadmin wykorzystuje tylko jeden port, na którym działa konsola administracyjna (przedstawiona na Rysunku 14).

Listing 8 - zawartość pliku Docker-Compose - definicja kontenerów NSQ

```
---
version: '3'
services:
  lookup:
    image: nsqio/nsq:v1.2.1
    container_name: nsq-lookupd
    command: /nsqlookupd
    ports:
      - "9104:4160" # Lookup TCP Address
      - "9105:4161" # Lookup HTTP Address

  broker:
    image: nsqio/nsq:v1.2.1
    container_name: nsq-broker
    command: /nsqd --lookupd-tcp-address nsq-lookupd:4160
    depends_on:
      - lookup
    ports:
      - "9102:4150" # TCP Port
      - "9103:4151" # HTTP Port

  admin:
    image: nsqio/nsq:v1.2.1
    container_name: nsq-admin
    command: /nsqadmin --lookupd-http-address nsq-lookupd:4161
    depends_on:
      - lookup
    ports:
      - "9106:4171" # HTTP Port

networks:
  default:
    external:
      name: cluster-network-1
```

5.6. SQS

Kolejkę wiadomości tworzy się i konfiguruje poprzez konsolę AWS – wystarczy przeglądarka internetowa. W konsoli należy wybrać zasób: Amazon SQS, co spowoduje przeniesienie użytkownika do listy kolejek (przedstawionej na Rysunku 16). Przycisk „Create queue” przenosi do formularza utworzenia nowej kolejki (Rysunek 15). Poza nazwą i typem (Standard/FIFO) opcjonalnie można dokonać dalszej konfiguracji takich parametrów jak czas retencji, maksymalny rozmiar wiadomości czy włączyć lub wyłączyć szyfrowanie.

Create queue

Details

Type
Choose the queue type for your application or cloud infrastructure.

☒ **Standard Info**
At-least-once delivery, message ordering isn't preserved

- At-least once delivery
- Best-effort ordering

☐ **FIFO Info**
First-in-first-out delivery, message ordering is preserved

- First-in-first-out delivery
- Exactly-once processing

You can't change the queue type after you create a queue.

Name
test-queue-1

A queue name is case-sensitive and can have up to 80 characters. You can use alphanumeric characters, hyphens (-), and underscores (_).

Standard queue

Standard queues support a nearly unlimited number of transactions per second (TPS) per API action (SendMessage, ReceiveMessage, or DeleteMessage). A message is delivered at least once, but occasionally more than one copy of a message is delivered. Standard queues provide best-effort ordering. Occasionally, messages might be delivered in an order different from which they were sent.

Standard queues are useful when very high throughput is important. Examples include:

- Decouple live user requests from intensive background work: let users upload media while resizing or encoding it.
- Allocate tasks to multiple worker nodes: process a high number of credit card validation requests.

Rysunek 15 - ekran konsoli AWS - formularz tworzenia nowej kolejki.

Po utworzeniu, nowa kolejka jest od razu widoczna na liście wszystkich kolejek i gotowa do użytku. Z poziomu konsoli można wykonać na niej kilka operacji, takich jak ręczne wysłanie lub odczytanie wiadomości, usunięcie wszystkich zapisanych komunikatów, czy ponowną edycję konfiguracji z pierwszego formularza – poza nazwą i typem kolejki, tych zmienić nie można.

Queues (1)

Search queues by prefix

Name	Type	Created	Messages available	Messages in flight	Encryption	Content-based deduplication
test-queue-1	Standard	Jan 14, 2023, 14:02:06 GMT+1	0	0	Amazon SQS key (SSE-SQS)	-
test-s24067-sqs-standard	Standard	Sep 28, 2022, 18:07:01 GMT+2	0	0	Amazon SQS key (SSE-SQS)	-

Rysunek 16 - ekran konsoli AWS - lista przypisanych do konta kolejek.

6. Implementacja aplikacji klienckich

Wybraną technologią jest Java w wersji 8 wraz ze szkieletem Spring Boot. Niewielką część kodu (klasy metryk Prometheusa) napisano w języku Kotlin. Tam, gdzie było to możliwe, skorzystano z oficjalnych bibliotek klienckich dla poszczególnych systemów kolejkowych.

Pisanie aplikacji klienckich rozpoczęto od klasy abstrakcyjnej `ProducerController`. Producent każdego z systemów kolejkowych rozszerza tę klasę, która łączy w sobie funkcjonalność nadawcy wiadomości i serwera HTTP API, służącego do sterowania działaniem komponentu. Kod klasy abstrakcyjnej przedstawiono na Listingu 9. Deklaruje ona metodę `produceMessages()` i implementuje `produceMessagesContinuously()` (która pierwszą metodę wzbogaca m. in. o wielowątkowość).

Listing 9 - kod generycznej klasy `ProducerController`. Listę importów i metody `get/set` pominięto

```
public abstract class ProducerController {

    private long messagesSentTotal = 0;
    protected Random random = new Random();
    private Map<String, Long> messagesSent = new HashMap<>();
    private Logger log = LoggerFactory.getLogger(this.getClass().getName());

    @PostMapping("/produce")
    public abstract void produceMessages(@RequestParam("count")
                                         Optional<Integer> countOptional,
                                         @RequestParam("message")
                                         Optional<String> messageOptional);

    @PostMapping("/produce-continuously")
    public void produceMessagesContinuously(@RequestParam("length")
                                             Optional<Integer> messageLength,
                                             @RequestParam("threads")
                                             Optional<Integer> threadCountOptional) {

        int threadCount = threadCountOptional.orElse(1);
        if(threadCount <= 0 || threadCount > 100) threadCount = 1;

        for(int i = 0; i < threadCount; i++) {
            new Thread(() -> {
                while(true) {
                    byte[] randomBytes = new byte[messageLength.orElse(10)];
                    random.nextBytes(randomBytes);
                    String randomMessage = new String(randomBytes, Charsets.UTF_8);
                    produceMessages(Optional.of(1), Optional.of(randomMessage));
                }
            }).start();
        }
    }

    protected synchronized void incrementCounter(String topic) {
        if(messagesSent.get(topic) == null)
            messagesSent.put(topic, new Long(1));
        else
            messagesSent.put(topic, messagesSent.get(topic) + 1);
        this.messagesSentTotal++;
    }

    public String getProducerType() {
        return "generic";
    }
}
```

Zmienne `messagesSentTotal`, `messagesSent` oraz metody `incrementCounter()` i `getProducerType()` służą do monitorowania – są wykorzystywane przez wbudowany eksporter metryk, który szerzej opisano w rozdziale 7.3.7.

Na potrzeby implementacji konsumentów również zaimplementowano klasę `GenericConsumer`, jednak z uwagi na to, że konsumentami nie trzeba sterować, zaimplementowano w niej tylko obsługę zliczania odczytanych wiadomości – analogicznie do `ProducerController`.

6.1. Kafka

Do implementacji producenta wykorzystano bibliotekę `kafka-clients` w wersji 3.3.1 (3 października 2022) oraz oficjalną dokumentację ([22]).

6.1.1. Producent

Najprostsza implementacja producenta polega na utworzeniu obiektu klasy `KafkaProducer`, a następnie zawinięciu (ang. *wrap*) wiadomości w obiekt klasy `ProducerRecord` (poza wiadomością zawierający takie metadane jak nazwa docelowego *topica*, opcjonalnie numer partycji czy nagłówki) i wysłaniu go przy użyciu `KafkaProducer`a. Listing 10 zawiera przykład prostego kodu wysyłającego wiadomość na *topic* `Kafka`. W podanym przykładzie kluczem wiadomości jest *epoch* w chwili jej wysłania, a jej wartością komunikat tekstowy.

Listing 10 - fragment kodu realizujący wysłanie wiadomości na topic Kafka

```
// Konfiguracja producenta
Properties properties = new Properties();
properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
    appConfig.getKafka().getBootstrapServer());
properties.put(ProducerConfig.ACKS_CONFIG, appConfig.getKafka().getAcks());
// Deklaracja klas klucza i wartości
properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
    LongSerializer.class.getName());
properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
    StringSerializer.class.getName());

// Utworzenie obiektu KafkaProducer z przygotowaną wcześniej konfiguracją
KafkaProducer<Long, String> producer = new KafkaProducer<>(properties);

// Przygotowanie wiadomości
String message = "Hello world";
ProducerRecord<Long, String> record = new ProducerRecord<>(
    "test-topic-1",
    Instant.now().toEpochMilli(),
    message);
producer.send(record);
```

Rozszerzono klasę `ProducerController`, nadpisując metodę `produceMessages()` (i dodając obsługę metryk – wywołanie metody `incrementCounter()`). Przed wysłaniem pierwszej wiadomości komponent tworzy nowy obiekt klasy `KafkaProducer`, którego właściwości wcześniej przygotowano w metodzie `createKafkaConfigMap()`. Wysyłanie kolejnych wiadomości jest zrealizowane w bloku kodu *try-with-resources*, po zakończeniu którego na obiekcie `KafkaProducer` automatycznie wzywana jest metoda `close()`. Zamknięty w taki sposób obiekt nie może zostać ponownie wykorzystany, stąd dla każdej paczki wiadomości tworzony jest nowy. Kod klasy `KafkaProducerController` znajduje się w Listingu 11.

Listing 11 - kod klasy `KafkaProducerController`

```
@RestController
@RequestMapping("kafka")
@NoArgsConstructor
@ConditionalOnExpression("${kafka.enabled}")
public class KafkaProducerController extends ProducerController {

    @Autowired
    private AppConfig appConfig;
    private Logger log = LoggerFactory.getLogger(this.getClass().getName());
    private Properties kafkaConfigMap = createKafkaConfigMap();

    @Override
    @PostMapping("/produce")
    public void produceMessages(@RequestParam("count")
                               Optional<Integer> countOptional,
                               @RequestParam("message")
                               Optional<String> messageOptional) {
        int count = countOptional.isPresent() ? countOptional.get() : 1;
        // try-with-resources z każdorazowo tworzonym
        // nowym obiektem klasy KafkaProducer
        try(Producer<Long, String> producer =
            new KafkaProducer<>(kafkaConfigMap)) {
            for(int i = 1; i <= count; i++) {
                ProducerRecord<Long, String> record = new ProducerRecord<>(
                    appConfig.getKafka().getOutboundTopic(),
                    Instant.now().toEpochMilli(),
                    messageOptional.isPresent()
                        ? messageOptional.get()
                        : UUID.randomUUID().toString());
                // Obiekt RecordMetadata zawiera informację zwrotną
                // o partycji i offsecie, na które trafiła wiadomość
                RecordMetadata metadata = producer.send(record).get();
                log.debug("Sent message to partition {} offset {}",
                    metadata.partition(),
                    metadata.offset());
                super.incrementCounter(appConfig.getKafka().getOutboundTopic());
            }
            producer.flush();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private Properties createKafkaConfigMap() {
        Properties properties = new Properties();
        properties.put(
            ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
            appConfig.getKafka().getBootstrapServer()); // localhost:9094
        properties.put(
            ProducerConfig.ACKS_CONFIG,
            appConfig.getKafka().getAcks());
        // Serializers
        properties.put(
            ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            LongSerializer.class.getName());
        properties.put(
            ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class.getName());
        return properties;
    }

    @Override
    public String getProducerType() {
        return "kafka";
    }
}
```

6.1.2. Konsument

Dzięki integracji *frameworka* Java Spring z Kafką¹⁴, można skorzystać z implementacji prostszej niż w przypadku biblioteki kafka-clients. Polega na zdefiniowaniu dwóch fabryk: `ConsumerFactory` (zawierającej konfigurację konsumenta) i `KafkaListenerContainerFactory`.

Listing 12 - definicje beanów fabryk `ConsumerFactory` i `KafkaListenerContainerFactory`

```
@Bean
public ConsumerFactory<String, String> kafkaConsumerFactory() {
    // Właściwości konsumentów
    Map<String, Object> props = new HashMap<>();
    props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
        appConfig.getKafka().getBootstrapServer());
    props.put(ConsumerConfig.GROUP_ID_CONFIG,
        appConfig.getKafka().getConsumerGroup());
    props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
        LongDeserializer.class);
    props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
        StringDeserializer.class);
    props.put(ConsumerConfig.PARTITION_ASSIGNMENT_STRATEGY_CONFIG,
        RoundRobinAssignor.class.getName());
    props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
        "earliest");
    return new DefaultKafkaConsumerFactory<>(props);
}

@Bean
public ConcurrentKafkaListenerContainerFactory<String, String>
kafkaListenerContainerFactory() {
    ConcurrentKafkaListenerContainerFactory<String, String> factory =
        new ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(kafkaConsumerFactory());
    return factory;
}
```

Po zdefiniowaniu *beanów*, wystarczy zaimplementować metody opisane adnotacją `@KafkaListener` w klasie `KafkaConsumer` (przedstawiona na Listingu 14). Metoda `incrementCounter()`, analogicznie do producenta, służy do zliczania odczytanych wiadomości na potrzeby monitorowania.

¹⁴ Dokumentacja dostępna jest pod linkiem: <https://spring.io/projects/spring-kafka> (dostęp 2022-12-10)

Listing 13 - kod klasy KafkaConsumer

```
@Component
@ConditionalOnProperty(value = "kafka.enabled", havingValue = "true")
public class KafkaConsumer extends GenericConsumer{

    @Autowired
    private AppConfig appConfig;
    private Logger log = LoggerFactory.getLogger(this.getClass().getName());

    @KafkaListener(topics="${kafka.inboundTopic}")
    public void consume(String message) {
        log.info(message);
        incrementCounter(appConfig.getKafka().getInboundTopic());
    }

    @Override
    public String getConsumerType() {
        return "kafka";
    }
}
```

6.1.3. Integracja z Apache Camel

Poza typową implementacją producenta i konsumenta, możliwe jest też zdefiniowanie *Camel Route*'a zaczynającego się i/lub kończącego się na *topicu* Kafki. *Route* w Listingu 6 definiuje przepływ zaczynający się od skonsumowania wiadomości z *topica* „input-topic-1”, zalogowaniu jej wraz z nagłówkami, a następnie wysłaniu jej na „output-topic-1”.

Listing 14 - prosty przepływ napisany w szkieletcie Camel

```
public class KafkaRouteBuilder extends RouteBuilder {

    private Logger log = LoggerFactory.getLogger(this.getClass().getName());

    @Override
    public void configure() throws Exception {
        from("kafka:localhost:9094?topic=input-topic-1")
            .routeId("SimpleKafkaRoute")
            .log(LoggingLevel.INFO, "Body: ${body}")
            .log(LoggingLevel.INFO, "Headers: ${headers}")
            .to("kafka:localhost:9094?topic=output-topic-1");
    }
}
```

6.1.4. Kafka Streams

Kafka Streams to *framework* pozwalający przetwarzać dane z *topiców* Kafki strumieniowo. Wiadomości są przetwarzane według raz zdefiniowanej topologii (analogicznie do Camel Route).

Przykład użycia Kafka Streams przedstawiono na Listingu 15. Po zdefiniowaniu wymaganej konfiguracji zdefiniowano topologię, która rozpoczyna się od *topica* wejściowego („inbound-topic”), a następnie rozdziela główny strumień na dwa podstrumienie:

Listing 15 - kod klasy *KafkaStreamsBuilder*, definiującej topologię i tworzącej *KafkaStreams*

```
@Component
@ConditionalOnProperty(value = "kafka.streamsEnabled", havingValue = "true")
public class KafkaStreamsBuilder {

    public KafkaStreamsBuilder(AppConfig appConfig) {
        // Konfiguracja właściwości klienta KafkaStreams
        Properties streamsProperties = new Properties();
        // Obowiązkowy identyfikator aplikacji
        streamsProperties.put(
            StreamsConfig.APPLICATION_ID_CONFIG,
            "Subscriber-Application-" + UUID.randomUUID());
        // Podanie adresu brokera Kafki
        streamsProperties.put(
            StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
            appConfig.getKafka().getBootstrapServer()); // "localhost:9094"
        // Zadeklarowanie klas obiektów klucza i wiadomości
        streamsProperties.put(
            StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
            Serdes.Long().getClass().getName());
        streamsProperties.put(
            StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
            Serdes.String().getClass().getName());

        String inboundTopic =
            appConfig.getKafka().getInboundTopic(); // "inbound-topic"
        String outboundTopic =
            appConfig.getKafka().getOutboundTopic(); // "outbound-topic"

        // Utworzenie nowego obiektu klasy StreamsBuilder,
        // służącego do generowania topologii
        StreamsBuilder builder = new StreamsBuilder();
        // Rozpoczęcie generowanie topologii na podstawie topica wejściowego
        builder
            .stream(inboundTopic, Consumed.with(Serdes.Long(), Serdes.String()))
            // Zdefiniowanie odgałęzienia,
            // do którego trafiają wiadomości zaczynające się od cyfry
            .split()
            .branch((k, v) -> v.matches("^([0-9]).+$"),
                Branched.withConsumer(branchStream1 ->
                    branchStream1
                        // Usunięcie wszystkich liter
                        // z wiadomości w tym odgałęzieniu
                        .mapValues(val ->
                            val.replaceAll("[a-zA-Z]", ""))
                        // Wysłanie zmodyfikowanych wiadomości na topic
                        // outbound-topic-1
                        .to(outboundTopic + "-1"))
                // Zdefiniowanie odgałęzienia,
                // do którego trafiają wiadomości zaczynające się od litery
                .branch((k, v) -> v.matches("^[a-zA-Z].+$"),
                    Branched.withConsumer(branchStream2 ->
                        branchStream2
                            // Usunięcie wszystkich cyfr
                            // z wiadomości w tym odgałęzieniu
                            .mapValues(val ->
                                val.replaceAll("[0-9]", ""))
                            // Wysłanie zmodyfikowanych wiadomości na topic
                            // outbound-topic-2
                            .to(outboundTopic + "-2"))));
        // Utworzenie i wystartowanie KafkaStreams na podstawie
        // zdefiniowanej topologii i właściwości
        KafkaStreams streams = new KafkaStreams(builder.build(), streamsProperties);
        streams.start();
    }
}
```

- do pierwszego podstrumienia trafiają wiadomości rozpoczynające się od litery, następnie są z nich usuwane wszystkie cyfry i powstała wiadomość trafia do *topica* wejściowego „outbound-topic-1”,
- do drugiego podstrumienia trafiają wiadomości rozpoczynające się od cyfry, po czym są z nich usuwane wszystkie litery, a powstała wiadomość jest wysyłana na „outbound-topic-2”.

6.2. RabbitMQ

W przypadku RabbitMQ zaimplementowano dwa zestawy producentów i konsumentów – obsługujące standardowe kolejki *point-to-point* i wzorzec *Pub/Sub*. Główna różnica pomiędzy obiema implementacjami polega na zadeklarowaniu celu/źródła w obiekcie klasy *Channel* – w przypadku *point-to-point* potrzebna jest nazwa kolejki, a *Pub/Sub* wymaga podania nazwy wymiany (ang. *exchange*) i klucza routującego.

Biblioteka, z których skorzystano, to oficjalna biblioteka kliencka RabbitMQ, implementująca protokół AMQP 0-9-1 – `com.rabbitmq:amqp-client:5.14.2` (luty 2022). Posłużono się oficjalną dokumentacją – [27].

6.2.1. Producent *point-to-point*

Obie implementacje wymagają utworzenia fabryki *ConnectionFactory* (Listing 16 – dla ułatwienia fabrykę utworzono w formie *beana*, aby móc skorzystać ze wstrzyknięcia zależności w obu producentach).

Listing 16 - definicja *beana* *ConnectionFactory*

```
@Bean
public ConnectionFactory rabbitConnectionFactory() {
    ConnectionFactory rabbitFactory = new ConnectionFactory();
    rabbitFactory.setHost(appConfig.getRabbitmq().getServer().getHostname());
    rabbitFactory.setPort(appConfig.getRabbitmq().getServer().getPort());
    return rabbitFactory;
}
```

Wysłanie wiadomości na kolejkę wymaga połączenia (*Connection*) i kanału (*Channel*). Połączenie jest produkowane przez fabrykę, a kanał tworzony przez połączenie. Kanał deklaruje docelową kolejkę (lub wymianę i klucz routowania – w przypadku *Pub/Sub*) wraz z zestawem jej parametrów – flag. Flagi są opcjonalne, jeżeli nie zostaną podane, kolejka będzie miała domyślne właściwości. Co ciekawe, domyślna kolejka zadeklarowana przy użyciu tej biblioteki nie jest trwała (*durable*) – a więc wiadomości wysłane na nią pod nieobecność konsumentów przepadają.

Listing 17 zawiera kod metody `createConnectionAndChannel()`, w której tworzone i przypisywane do zmiennych instancji są połączenie i kanał, a następnie jest deklарowana kolejka.

Listing 17 - fragment kodu klasy *RabbitMQProducerController* - metoda *createConnectionAndChannel()*

```
private void createConnectionAndChannel() throws IOException, TimeoutException {
    // O ile połączenie nie istnieje i nie jest otwarte, wyprodukowanie nowego z fabryki
    if(rabbitConnection == null || !rabbitConnection.isOpen())
        rabbitConnection = rabbitFactory.newConnection();
    // Analogicznie w przypadku kanału
    if(rabbitChannel == null || !rabbitChannel.isOpen()) {
        rabbitChannel = rabbitConnection.createChannel();
        durableMessages = appConfig.getRabbitmq().getMsq().isDurableMessages();
        exclusiveQueue = appConfig.getRabbitmq().getMsq().isExclusiveQueue();
        // Deklaracja kolejki o właściwościach zdefiniowanych
        // w konfiguracji aplikacji
        rabbitChannel.queueDeclare(
            appConfig.getRabbitmq().getMsq().getOutboundQueue(),
            durableMessages,
            exclusiveQueue,
            !durableMessages,
            null);
    }
}
```

Na Listingu 18 przedstawiono implementację wysyłania wiadomości w modelu *point-to-point*. Przed wysłaniem każdej paczki wiadomości, wywoływana jest metoda *createConnectionAndChannel()*. Kiedy już upewniono się, że połączenie i kanał deklarujący odpowiednią kolejkę istnieją, wywoływana jest metoda *basicPublish()*. Ta sama metoda służy też do wysyłania wiadomości w modelu *Pub/Sub*, w tym przypadku po prostu nazwa wymiany pozostaje pusta, a klucz routingu to nazwa docelowej kolejki.

Listing 18 - fragment kodu klasy *RabbitMQProducerController* - metoda *produceMessages()*

```
@Override
@PostMapping("/produce")
public void produceMessages(@RequestParam("count")
    Optional<Integer> countOptional,
    @RequestParam("message")
    Optional<String> messageOptional) {
    int count = countOptional.orElse(1);
    try{
        createConnectionAndChannel();
        for(int i = 1; i <= count; i++) {
            rabbitChannel.basicPublish(
                Strings.EMPTY,
                appConfig.getRabbitmq().getMsq().getOutboundQueue(),
                // Jeżeli wiadomości mają być trwałe,
                // ustawiany jest odpowiedni gotowy zestaw właściwości
                durableMessages
                    ? MessageProperties.MINIMAL_PERSISTENT_BASIC
                    : null,
                messageOptional.isPresent()
                    ? messageOptional.get().getBytes()
                    : UUID.randomUUID().toString().getBytes()
            );
            super.incrementCounter(appConfig.getRabbitmq().getMsq().getOutboundQueue());
        }
    } catch(Exception e) {
        e.printStackTrace();
    }
}
```

6.2.2. Producent *Pub/Sub*

Implementacja modelu *Pub/Sub* jest podobna do *point-to-point*, poniżej opisano główne różnice:

- klasa implementująca *Pub/Sub*, `RabbitPubSubProducerController`, również ma metodę `createConnectionAndChannel()`, ale utworzony kanał, zamiast deklarować kolejkę, deklaruje wymianę (Listing 19),

Listing 19 - fragment kodu metody `createConnectionAndChannel()` klasy `RabbitPubSubProducerController`

```
rabbitChannel.exchangeDeclare(  
    rabbitExchange,  
    rabbitRoutingKey,  
    durableMessages);
```

- w `produceMessages()` metoda `basicPublish()` została wezwana z innymi parametrami (Listing 20).

Listing 20 - fragment kodu metody `produceMessages()` klasy `RabbitPubSubProducerController`

```
rabbitChannel.basicPublish(  
    rabbitExchange,  
    rabbitRoutingKey, // ""  
    durableMessages  
        ? MessageProperties.PERSISTENT_TEXT_PLAIN  
        : null,  
    messageOptional.isPresent()  
        ? messageOptional.get().getBytes()  
        : UUID.randomUUID().toString().getBytes());
```

Listing 21 - kod klasy *RabbitMQConsumer*

```
@Component
@ConditionalOnProperty(value = "rabbitmq.msq.enabled", havingValue = "true")
public class RabbitMQConsumer extends GenericConsumer {

    private AppConfig appConfig;
    private ConnectionFactory rabbitConnectionFactory;
    private Logger log = LoggerFactory.getLogger(this.getClass().getName());
    private String rabbitQueue;
    private boolean durableMessages;
    private boolean exclusiveQueue;

    public RabbitMQConsumer(AppConfig appConfig,
                           ConnectionFactory rabbitConnectionFactory) {
        this.appConfig = appConfig;
        this.rabbitConnectionFactory = rabbitConnectionFactory;
        this.rabbitQueue = appConfig.getRabbitmq().getMsq().getInboundQueue();
        this.durableMessages = appConfig.getRabbitmq().getMsq().isDurableMessages();
        this.exclusiveQueue = appConfig.getRabbitmq().getMsq().isExclusiveQueue();
        initConsumer();
    }

    private boolean initConsumer() {
        try {
            Connection rabbitConnection = rabbitConnectionFactory.newConnection();
            Channel rabbitChannel = rabbitConnection.createChannel();
            // Deklaracja kolejki, z której Consumer będzie czytał
            rabbitChannel.queueDeclare(
                rabbitQueue, // queueName
                durableMessages, // durable
                exclusiveQueue, // exclusive
                !durableMessages, // autoDelete
                null); // arguments

            // Zdefiniowanie funkcji lambda wyzwalanej dla każdej
            // odebranej wiadomości
            DeliverCallback messageCallback = (tag, payload) -> {
                String message = new String(payload.getBody(), "UTF-8");
                log.info(message);
                incrementCounter(rabbitQueue);
            };

            // Związanie kolejki z funkcją lambda
            rabbitChannel.basicConsume(
                rabbitQueue,
                true,
                messageCallback,
                (tag) -> {
                });

        } catch (Exception e) {
            return false;
        }
        return true;
    }

    @Override
    public String getConsumerType() {
        return "rabbitmq/msq";
    }
}
```

6.2.3. Konsument point-to-point

Aplikacja konsumenta również składa się z dwóch implementacji. Obie korzystają z tej samej biblioteki co producent i też wymagają utworzenia fabryki `ConnectionFactory` (Listing 16). Na Listingu 21 przedstawiony został kod klasy `RabbitMQConsumer`, realizującej czytanie wiadomości w modelu *point-to-point*. W metodzie `initConsumer()`:

- utworzono połączenie i kanał,
- zadeklarowano kolejkę na podstawie konfiguracji aplikacji,
- zdefiniowano funkcję lambda, wyzwalaną po odebraniu każdej wiadomości – w tym wypadku logującej komunikat i inkrementującej licznik (metrykę)
- związano funkcję lambda z kolejką, rozpoczynając subskrypcję.

6.2.4. Konsument Pub/Sub

Listing 22 - fragment kodu klasy `RabbitMQSubscriber` - metoda `initConsumer()`

```
private boolean initConsumer() {
    try {
        Connection rabbitConnection = rabbitConnectionFactory.newConnection();
        Channel rabbitChannel = rabbitConnection.createChannel();

        rabbitChannel.exchangeDeclare(
            rabbitExchange,
            Strings.EMPTY,
            durableMessages);

        if(namedQueue)
            rabbitQueue = rabbitChannel.queueDeclare(
                queuePrefix + rabbitExchange, // queueName
                durableMessages, // durable
                exclusiveQueue, // exclusive
                false, // autoDelete
                null).getQueue(); // arguments
        else
            rabbitQueue = rabbitChannel.queueDeclare().getQueue();

        rabbitChannel.queueBind(
            rabbitQueue,
            rabbitExchange,
            rabbitBindingKey);

        DeliverCallback messageCallback = (tag, payload) -> {
            String message = new String(payload.getBody(), "UTF-8");
            log.info(message);
            incrementCounter(rabbitExchange);
        };

        rabbitChannel.basicConsume(
            rabbitQueue,
            true,
            messageCallback,
            (tag) -> {});
    }
    catch(Exception e) {
        e.printStackTrace();
        return false;
    }
    return true;
}
```

Konsument *Pub/Sub* różni się od *point-to-point* metodą `initConsumer()`. Kod metody przedstawiony jest na Listingu 22. Utworzony kanał deklaruje jednocześnie wymianę i kolejkę, a następnie

związuje kolejkę z wymianą. Następnie, analogicznie jak w poprzednim przypadku, definiowana jest funkcja lambda, po czym jest ona związywana z kolejką.

6.3. RocketMQ

RocketMQ dysponuje oficjalną biblioteką kliencką Java przeznaczoną dla szkieletu Spring Boot. Opis implementacji prostego producenta i konsumenta przedstawiony został w [3].

Zdecydowano się jednak skorzystać z przykładu w oficjalnej dokumentacji RocketMQ (patrz: [32]), korzystając z biblioteki org.apache.rocketmq.rocketmq-client:4.9.3 (marzec 2022).

6.3.1. Producent

Listing 23 - fragment kodu klasy `RocketMQProducerController` - przeciążone metody `produceMessages()`

```
@Override
@PostMapping("/produce")
public void produceMessages(@RequestParam("count")
                           Optional<Integer> countOptional,
                           @RequestParam("message")
                           Optional<String> messageOptional) {
    produceMessages(countOptional, Optional.of(Strings.EMPTY), messageOptional);
}

@PostMapping("/produce-with-tag")
public void produceMessages(@RequestParam("count")
                           Optional<Integer> countOptional,
                           @RequestParam("tag")
                           Optional<String> tagOptional,
                           @RequestParam("message")
                           Optional<String> messageOptional) {
    int count = countOptional.orElse(1);

    try{
        // Utworzenie domyślnego obiektu Producenta, deklaracja adresu nameservera
        if(rocketProducer == null) {
            rocketProducer = new DefaultMQProducer();
            rocketProducer.setNamesrvAddr(
                appConfig.getRocketmq().getNameserver());
            rocketProducer.setProducerGroup(
                appConfig.getRocketmq().getGroupName());
            rocketProducer.start();
        }

        for(int i = 1; i <= count; i++) {
            // Utworzenie wiadomości
            Message message = new Message(
                appConfig.getRocketmq().getOutboundTopic(),
                tagOptional.orElse(Strings.EMPTY),
                messageOptional.orElse(UUID.randomUUID().toString()).getBytes());
            // Wysłanie na topic zadeklarowany w wiadomości.
            // Obiekt response zawiera metadane, takie jak status wysyłki,
            // docelowy offset albo kolejka.
            SendResult response = rocketProducer.send(message);
            super.incrementCounter(appConfig.getRocketmq().getOutboundTopic());
        }
    } catch(Exception e) {
        e.printStackTrace();
    }
}
```

Zaimplementowany producent wysyła wiadomości synchronicznie (patrz: 4.4.). Fragment kodu klasy `RocketMQProducerController` przedstawiono na Listingu 23. RocketMQ pozwala opatrzyć wysyłaną wiadomość etykietą (ang. *tag*), więc standardową metodę `produceMessages()` przeciążono, dodając sygnaturę umożliwiającą wysłanie wiadomości z etykietą.

Co ciekawe, producent wymaga ustawienia nazwy grupy producenckiej (ang. *producer group*) – próba uruchomienia komponentu bez tego parametru kończy się rzuceniem wyjątku. W Internecie nie znaleziono informacji na temat grup producenckich RocketMQ. Ustawienie dowolnej wartości tekstowej rozwiązuje problem. Być może jest to pozostałość po starej, porzuconej funkcjonalności.

Obiekt producenta nadaje się do ponownego użycia, ponieważ nazwa docelowego *topica* jest atrybutem wiadomości.

6.3.2. Konsument

Są dwie metody konsumowania wiadomości, zdefiniowane w dokumentacji RocketMQ: *Push* i *Pull* (ta druga dzieli się jeszcze na standardową i *Lite*). Jak opisano w [30], *Push* jest metodą bezinicjatywową – klient musi tylko zasubskrybować *topic* i grupę, a broker sam będzie dostarczał wiadomości, wyzwalając zdefiniowaną przez konsumenta procedurę obsługi wiadomości (*callback*). Takie rozwiązanie jest wydajniejsze, ale w przypadku dużej liczby odebranych wiadomości, może doprowadzić do utopienia (ang. *flood*) klienta i w konsekwencji jego awarii.

Listing 24 - fragment kodu klasy `RocketMQConsumer` - metoda `initConsumer()`

```
private void initConsumer() {
    try {
        // Utworzenie obiektu konsumenta, podanie adresu nameservera,
        // deklaracja nazwy grupy i topica
        DefaultMQPushConsumer rocketConsumer = new DefaultMQPushConsumer(
            appConfig.getRocketmq().getGroupName()); // consumer group name
        rocketConsumer.setNamesrvAddr(
            appConfig.getRocketmq().getNameserver());
        rocketConsumer.subscribe(
            appConfig.getRocketmq().getInboundTopic(), // topic name
            "*"); // sub-expression

        // Definicja funkcji wyzwalanej przez odbiór wiadomości,
        // powiązanie jej z konsumentem
        rocketConsumer.registerMessageListener(
            (MessageListenerConcurrently) (messages, context) -> {
                messages.stream()
                    .forEach(msg -> {
                        log.info("Received message: {}",
                                new String(msg.getBody()));
                        incrementCounter(
                            appConfig.getRocketmq().getInboundTopic());
                    });
                return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
            });

        // Rozpoczęcie subskrypcji
        rocketConsumer.start();
        log.info("Consumer started");
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
```

Pull jest odwrotnym podejściem – konsument, kiedy jest gotowy, wysyła zapytanie o nowe wiadomości do brokera. Przenosi to ciężar z klienta na serwer, który musi obsłużyć częste zapytania wysyłane przez jednego lub kilku konsumentów.

Zaimplementowano konsumenta korzystającego z metody *Push*. Utworzony obiekt klasy `DefaultMQPushConsumer` deklaruje adres *nameservera*, grupy i *topica*. Definiowana jest metoda obsługująca otrzymanie wiadomości, a następnie jest wiązana z obiektem konsumenta. Wszystkie te kroki są realizowane przez metodę `initConsumer()`, której kod znajduje się w Listingu 24.

6.4. ActiveMQ

Skorzystano z oficjalnej biblioteki Java w wersji 5.16.5 (maj 2022) i przykładu z oficjalnej dokumentacji – [13]. Zarówno producent jak i konsument definiują fabrykę połączeń do serwera – obiekt klasy `ActiveMQConnectionFactory` (Listing 25).

*Listing 25 - fragment kodu aplikacji producenta - definicja beana
activeConnectionFactory*

```
@Bean
public ActiveMQConnectionFactory activeConnectionFactory() {
    return new ActiveMQConnectionFactory(
        String.format("tcp://%s", appConfig.getActivemq().getServer())
    );
}
```

6.4.1. Producent

Poza utworzeniem fabryki cały proces wysyłania wiadomości jest realizowany przez standardową metodę `produceMessages()` (Listing 26). Wymaga utworzenia obiektów połączenia, sesji, celu (ang. *destination*) i producenta, a dopiero wtedy można wysłać na kolejkę wiadomość. W porównaniu do implementacji producentów np. Kafki albo RocketMQ, proces jest skomplikowany.

Obiekt klasy `MessageProducer` pozwala na ustawienie jednego z dwóch trybów dostarczenia wiadomości: persystentnego lub niepersystentnego. Różnica pomiędzy nimi została wyjaśniona w oficjalnej dokumentacji ActiveMQ (patrz: [14] i [18]). Persystentny tryb oznacza, że wiadomości są zapisywane na trwałym nośniku na wypadek awarii lub restartu brokera. Pojęcia *persistence* i *durability* są tu więc używane naprzemiennie, co nie zgadza się z definicjami przedstawionymi w [39].

Listing 26 - fragment kodu klasy `ActiveMQProducerController` - metoda `produceMessages()`

```
@Override
@PostMapping("/produce")
public void produceMessages(@RequestParam("count")
                           Optional<Integer> countOptional,
                           @RequestParam("message")
                           Optional<String> messageOptional) {

    try {
        // Wyprodukowanie nowego połączenia z fabryki i nazwiązanie go
        Connection activeConnection = activeConnectionFactory.createConnection();
        activeConnection.start();

        // Utworzenie sesji
        Session activeSession = activeConnection.createSession(
            false, // transacted
            Session.AUTO_ACKNOWLEDGE); // acknowledge mode

        // Zdefiniowanie celu - w tym przypadku kolejki
        Destination outboundQueue = activeSession.createQueue(
            appConfig.getActivemq().getOutboundQueue()); // queue name

        // Utworzenie producenta, powiązanie go z celem i ustawienie właściwości
        MessageProducer activeProducer = activeSession.createProducer(outboundQueue);
        activeProducer.setDeliveryMode(DeliveryMode.PERSISTENT);

        for(int i = 1; i <= countOptional.orElse(1); i++) {
            // Wysłanie wiadomości tekstowej przy użyciu producenta
            activeProducer.send(
                activeSession.createTextMessage(
                    messageOptional.orElse(UUID.randomUUID().toString())
                )
            );
            super.incrementCounter(appConfig.getActivemq().getOutboundQueue());
        }

        // Zamknięcie sesji i połączenia
        activeSession.close();
        activeConnection.close();

    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

6.4.2. Konsument

Odczytywanie wiadomości, podobnie jak w przypadku poprzednich implementacji, realizowane jest metodą *Push* – za pomocą nasłuchiwanie wiadomości (ang. *message listener*).

Ponownie tworzone są połączenie i sesja, ale tym razem zamiast obiektu producenta tworzony jest konsument, deklarujący nazwę kolejki. Następnie definiowana jest funkcja obsługująca odebrane wiadomości i przypisywana do konsumenta. W kodzie aplikacji przedstawionym na Listingu 27, ostatnie dwa kroki zostały zrealizowane równocześnie, dzięki zastosowaniu funkcji lambda.

Listing 27 – fragment kodu klasy *ActiveMQConsumer* - metoda *initConsumer()*

```
private boolean initConsumer() {
    try {
        Connection activeConnection = activeConnectionFactory.createConnection();
        activeConnection.start();

        Session activeSession = activeConnection.createSession(
            false, // transacted
            Session.CLIENT_ACKNOWLEDGE); // acknowledge mode

        MessageConsumer activeConsumer =
            activeSession.createConsumer(
                activeSession.createQueue(
                    appConfig.getActivemq().getInboundQueue()));

        // Definicja i ustawienie funkcji obsługującej odebrane wiadomości
        activeConsumer.setMessageListener(message -> {
            try {
                TextMessage textMessage = (TextMessage) message;
                log.info(textMessage.getText());
                incrementCounter(appConfig.getActivemq().getInboundQueue());
                message.acknowledge();
            } catch (Exception e) {
                e.printStackTrace();
            }
        });
        return true;
    } catch (Exception e) {
        e.printStackTrace();
        return false;
    }
}
```

Odebrana wiadomość przed zalogowaniem jest rzutowana na klasę *TextMessage*, pochodzącą z biblioteki klienckiej. Poza wiadomościami tekstowymi ActiveMQ wspiera też między innymi:

- BLOBy,
- mapy,
- obiekty,
- strumienie (ang. *stream*).

6.5. NSQ

Istnieje kilka bibliotek klienckich przeznaczonych dla języka Java¹⁵, ale żadna z nich nie jest oficjalna. Zdecydowano się wybrać *com.sproutsocial.nsq-j:1.2* (listopad 2020)¹⁶. Poniższe podrozdziały opisują implementację producenta i konsumenta z użyciem tej biblioteki.

6.5.1. Producent

Wybrana biblioteka umożliwia wysyłanie wiadomości w dwóch trybach: standardowym i buforowanym. W trybie buforowanym komunikaty są najpierw ładowane do bufora, a bufor wysyła je na

¹⁵ Lista bibliotek dla poszczególnych języków znajduje się na stronie oficjalnej dokumentacji – <https://nsq.io/clients/client-libraries.html> (dostęp 2023-01-15)

¹⁶ Strona projektu na GitHub: <https://github.com/sproutsocial/nsq-j> (dostęp 2023-01-15)

topic po wypełnieniu (domyślnie 16 kB) lub przekroczeniu ustalonego czasu (domyślnie 300 ms). Klasa producenta `NSQProducerController` implementuje metody realizujące oba sposoby – przedstawione na Listingu 28.

Listing 28 - fragment kodu klasy `NSQProducerController` - metody `produceMessage` i `produceMessageBuffered`

```
private void produceMessage(Publisher publisher, String message, String topic) {
    publisher.publish(
        topic,
        message.getBytes()
    );
    super.incrementCounter(topic);
}

private void produceMessageBuffered(Publisher publisher, String message, String topic) {
    publisher.publishBuffered(
        topic,
        message.getBytes()
    );
    super.incrementCounter(topic);
}
```

Co istotne, korzystając z biblioteki `nsq-j`, wysyłamy i odbieramy wiadomości w postaci wektora bajtów – kwestia ich parsowania i rzutowania leży w gestii umowy pomiędzy producentem i konsumentem.

Aby wysłać wiadomość na jeden z dwóch sposobów, należy utworzyć obiekt klasy `Publisher`, dostarczanej przez bibliotekę, a następnie przekazać go wraz z wiadomością i nazwą *topica* do jednej z powyższych metod (przykład na Listingu 29).

Listing 29 - przykład użycia metody z Listingu 28 - wysłanie wiadomości w sposób buforowany

```
...
Publisher publisher = new Publisher("localhost:9102"); // adres brokera
produceMessageBuffered(
    publisher,                                // utworzony Publisher
    "Test message!",                          // treść wiadomości
    "test-topic-1");                          // nazwa topica
...
```

6.5.2. Konsument

W przypadku `NSQ` zaimplementowano konsumenta pozwalającego się skalować wraz ze skonfigurowaną liczbą wątków. Kod klasy przedstawiono na Listingu 30. Każdy powołany wątek tworzy obiekt klasy `Subscriber` dostarczanej przez `nsq-j`, po czym subskrybuje *topic* i grupę. Wszystkie wątki korzystają z tej samej konfiguracji, a więc łączą się do tego samego serwera, grupy i *topica*.

Odebrane wiadomości mają postać wektora bajtów, przed zalogowaniem są konwertowane do postaci tekstowej (domyślnie UTF-8).

Listing 30 - kod klasy NSQConsumer

```
@Component
@ConditionalOnProperty(value = "nsq.enabled", havingValue = "true")
public class NSQConsumer extends GenericConsumer {

    private static AppConfig appConfig;
    private static Logger log;
    private static NSQConsumer instance;

    public NSQConsumer(AppConfig appConfig) {
        instance = this;
        this.appConfig = appConfig;
        log = LoggerFactory.getLogger(this.getClass().getName());

        // Utworzenie n wątków-konsumentów (domyślnie 1)
        for(int i = 0; i < appConfig.getNsq().getConsumerCount(); i++) {
            new Thread(() -> initConsumer()).start();
        }
    }

    private void initConsumer() {
        log.info("Starting consumer: {}", appConfig.getNsq());
        // utworzenie subskrybenta i zasubskrybowanie topica/kanału
        // z referencją do funkcji (statycznej metody) realizującej obsługę wiadomości
        new Subscriber(appConfig.getNsq().getServer())
            .subscribe(
                appConfig.getNsq().getInboundTopic(), // nazwa topica
                appConfig.getNsq().getChannel(),       // nazwa kanału
                NSQConsumer::handleData);              // referencja do funkcji
        log.info("Subscribed!");
    }

    private static void handleData(byte[] data) {
        log.info(new String(data));
        getInstance().incrementCounter(appConfig.getNsq().getInboundTopic());
    }

    @Override
    public String getConsumerType() {
        return "nsq";
    }

    private static NSQConsumer getInstance() {
        return instance;
    }
}
```

6.6. SQS

Implementacja aplikacji klienckich SQS różni się od pozostałych systemów kolejkowych. Klient każdorazowo łączy się do centrum serwerowego Amazona – musi więc wiedzieć, do którego regionu AWS musi nawiązać połączenie (wybrano eu-central-1 – Frankfurt). Nie ma też mowy o nieautoryzowanym połączeniu.

Jedną ze wspieranych przez AWS metod uwierzytelniania jest podanie kluczy dostępu. Klucz wraz z sekretem generuje się w konsoli AWS. Każdy posiadacz takich kluczy może uwierzytelnić się jako użytkownik AWS (IAM User) i o ile został mu przyznany, autoryzować dostęp do kolejki SQS.

Do napisania producenta i konsumenta użyto oficjalnych bibliotek klienckich Amazona: `com.amazonaws.aws-java-sdk-sqs` w wersji 1.12.307 (wrzesień 2022).

6.6.1. Producent

Listing 31 - fragment aplikacji producenta - definicja beana *amazonSqsClient*

```
@Bean
public AmazonSQS amazonSqsClient() {
    // zawinięcie klucza i sekretu z konfiguracji
    // w obiekt klasy BasicAWSCredentials
    BasicAWSCredentials credentials =
        new BasicAWSCredentials(
            appConfig.getSqs().getAccessKeyId(),
            appConfig.getSqs().getAccessKeySecret());

    // wybudowanie klienta SQS - podanie typu kolejki,
    // regionu AWS i uwierzytelniania
    return AmazonSQSClientBuilder
        .standard()
        .withRegion(appConfig.getSqs().getRegion())
        .withCredentials(new AWSStaticCredentialsProvider(credentials))
        .build();
}
```

Wysyłanie i odpytywanie o wiadomości są realizowane za pośrednictwem obiektu klasy AmazonSQS. Dla wygody utworzono taki obiekt jako *bean* (Listing 31). Wykorzystano budowniczego (ang. *builder*) i podano:

- typ kolejki,
- region AWS,
- klucze dostępu.

Listing 32 - fragment kodu klasy *SQSProducerController* – metoda *produceMessages()*

```
@Override
@PostMapping("/produce")
public void produceMessages(@RequestParam("count") Optional<Integer> countOptional,
    @RequestParam("message") Optional<String> messageOptional) {
    // znalezienie URL kolejki
    if(queueName == null) {
        queueName = appConfig.getSqs().getQueueName();
        queueUrl = amazonSqsClient.getQueueUrl(queueName).getQueueUrl();
    }
    // wysyłanie wiadomości w pętli
    for(int i = 1; i <= countOptional.orElse(1); i++) {
        amazonSqsClient.sendMessage(
            new SendMessageRequest(
                queueUrl,
                messageOptional.isPresent()
                    ? messageOptional.get()
                    : UUID.randomUUID().toString()
            )
        );
        super.incrementCounter(queueName);
    }
}
```

Dzięki adnotacji *@Autowired* obiekt został wstrzyknięty do *SQSProducerController*, gdzie posłużył do:

- odnalezienia URL kolejki na podstawie jej nazwy,
- wysłania na kolejkę komunikatu.

Fragment kodu przedstawiony w Listingu 32 realizuje powyższe kroki.

6.6.2. Konsument

Konsument został napisany z wykorzystaniem mechanizmu rzadkiego odpytywania (patrz: 4.6.) – z tego powodu poprzednią definicję beana `amazonSqsClient` rozszerzono o wydłużony maksymalny czas połączenia (ang. *timeout*), równy 30s.

W przeciwieństwie do pozostałych systemów kolejkowych, SQS nie umożliwia subskrypcji wiadomości metodą *Push*. Serwer jest odpytywany w nieskończonej pętli. W zależności od odpowiedzi:

- jeżeli serwer zwróci listę wiadomości, konsument procesuje je, dla każdej generując wpis na liście wiadomości do usunięcia. Po przetworzeniu wszystkich, wysyła zbiorcze żądanie usunięcia wiadomości,
- jeżeli odpowiedź serwera nie zawiera wiadomości, klient poczeka skonfigurowany interwał (w przykładzie na Listingu 33 jest to 20 sekund) zanim ponownie wyśle zapytanie. Jest to właśnie mechanizm *long-poll* mający na celu redukcję kosztów użytkowania kolejki.

*Listing 33 - fragment kodu klasy `SQSConsumer` - czytanie wiadomości metodą *long-poll**

```
...
// utworzenie szablonu żądania odpytania,
// zadeklarowanie URL kolejki i czas oczekiwania
// po "pustej" odpowiedzi (long-polling)
ReceiveMessageRequest receiveRequest = new ReceiveMessageRequest()
    .withQueueUrl(queueUrl)
    .withWaitTimeSeconds(20);

while(true) {
    log.info("Polling...");
    List<DeleteMessageBatchRequestEntry> messagesReceipts =
        amazonSqsClient
            // odpytanie kolejki
            .receiveMessage(receiveRequest)
            // strumieniowe przetwarzanie zwróconych wiadomości
            .getMessages()
            .stream()
            .peek(message -> {
                String data = message.getBody();
                log.info("Received: {}", data);
                incrementCounter(queueName);
            })
            // dla każdej odczytanej wiadomości utworzenie
            // żądania usunięcia
            .map(message ->
                new DeleteMessageBatchRequestEntry(
                    message.getMessageId(),
                    message.getReceiptHandle()))
            .collect(Collectors.toList());

    if(!messagesReceipts.isEmpty())
        // usunięcie odczytanych wcześniej wiadomości
        amazonSqsClient.deleteMessageBatch(
            new DeleteMessageBatchRequest(
                queueUrl,
                messagesReceipts));
}
...
```

7. Instalacja i konfiguracja narzędzi do monitorowania

Poniższe podrozdziały dotyczą technologii wykorzystanej do monitorowania systemów kolejkowych i aplikacji klienckiej. Pokróćce opisano szkielet monitorowania Prometheus i Grafanę oraz ich instalację i konfigurację. Do każdego z monitorowanych komponentów dobrano eksportery metryk – tam, gdzie było to możliwe skorzystano z oficjalnych rozwiązań lub popularnych projektów opracowanych przez społeczność, ale w przypadku NSQ zaimplementowano własne narzędzie przekształcające metryki do formatu Prometheusa. Opisano implementację wbudowanych eksporterów metryk mikroserwisów producenta i konsumenta.

7.1. Prometheus

Prometheus jest jednym z najpopularniejszych narzędzi do monitorowania systemów¹⁷. Początkowo został opracowany przez SoundCloud na własny użytek, a następnie przekształcony w projekt open-source, którego kod źródłowy jest udostępniony na GitHub [44][45].

Prometheus przechowuje metryki, czyli liczby opisujące pracę systemu. Te liczby są opisane zbiorem etykiet (ang. *labels*), które, przy wykorzystaniu zapytań języka PromQL, umożliwiają tworzenie wykresów, reguł alarmowania (ang. *alerts*). Listing 34 zawiera przykładowe metryki w formacie tekstowym. Monitorowane aplikacje (lub eksportery metryk) eksponują *endpointy* HTTP, które na zapytanie GET odpowiadają listą takich metryk. *Endpointy* są zapisane w konfiguracji Prometheusa, który zbiera z nich metryki co określony interwał i zapisuje w swojej bazie danych.

Listing 34 - przykładowe metryki wygenerowane przez eksporter Kafki. Wartości metryk odpowiadają offsetowi (łącznej liczbie otrzymanych wiadomości) na danej partycji danego topica Kafki

```
kafka_topic_partition_current_offset{partition="0",topic="test-topic-1"} 123
kafka_topic_partition_current_offset{partition="0",topic="test-topic-2"} 234
kafka_topic_partition_current_offset{partition="0",topic="test-topic-3"} 345
```

Definicję kontenera zawierającego serwer Prometheus przedstawiono w Listingu 35. Kontener bazuje na oficjalnym obrazie w wersji v2.34.0 (wydanej w marcu 2022). Flaga `--web.enable-lifecycle` dodana do komendy startowej jest opcjonalna i pozwala wyzwolić przeładowanie konfiguracji z pliku poprzez API, zamiast restartu kontenera (co jest czasochłonne, ponieważ po starcie Prometheus wczytuje cały odnaleziony w plikach log sekwencyjny WAL). Utworzono wolumen, aby pliki konfiguracyjne były dostępne bezpośrednio na maszynie, na której działa Docker, a także aby zachować dane po zrekreowaniu kontenera.

¹⁷ Prometheus zajmuje na przykład miejsce na podium w rankingu najpopularniejszych narzędzi do monitorowania Stackshare, obok Grafany (z którą wykorzystywany jest zazwyczaj w parze) i Kibana (która najczęściej występuje w tak zwanym *stacku EFK*, mającym inne zastosowanie) – <https://stackshare.io/monitoring-tools> (dostęp 2023-01-02)

Listing 35 - definicja kontenera z serwerem Prometheus napisana w Docker-Compose

```
---
version: '3'
services:
  prometheus:
    image: prom/prometheus:v2.34.0
    container_name: prometheus
    command: --web.enable-lifecycle
    ports:
      - "9090:9090"
    volumes:
      - "/opt/masters-configs/monitoring/prometheus:/prometheus"
networks:
  default:
    external:
      name: cluster-network-1
```

W dużej mierze wykorzystano domyślną konfigurację, ale konieczne jest zdefiniowanie listy punktów końcowych (ang. *endpoint*) eksporterów metryk, z których mają być zbierane dane. Część pliku konfiguracyjnego przedstawiono w Listingu 36. Komunikacja z eksporterami metryk odbywa się w obrębie sieci dockerowej, więc URLami mogą być nazwy kontenerów. W podanym przykładzie ścieżka punktu końcowego z metrykami RabbitMQ została podana, ponieważ jest inna niż domyślna („/metrics”).

Listing 36 - fragment zawartości pliku konfiguracyjnego Prometheusa

```
scrape_configs:
  - job_name: kafka
    scrape_interval: 10s
    scrape_timeout: 9s
    static_configs:
      - targets:
        - "kafka-exporter:9308"
  - job_name: rabbitmq
    scrape_interval: 10s
    scrape_timeout: 9s
    metrics_path: "/metrics/per-object"
    static_configs:
      - targets:
        - "rabbitmq:15692"
...
```

7.2. Grafana

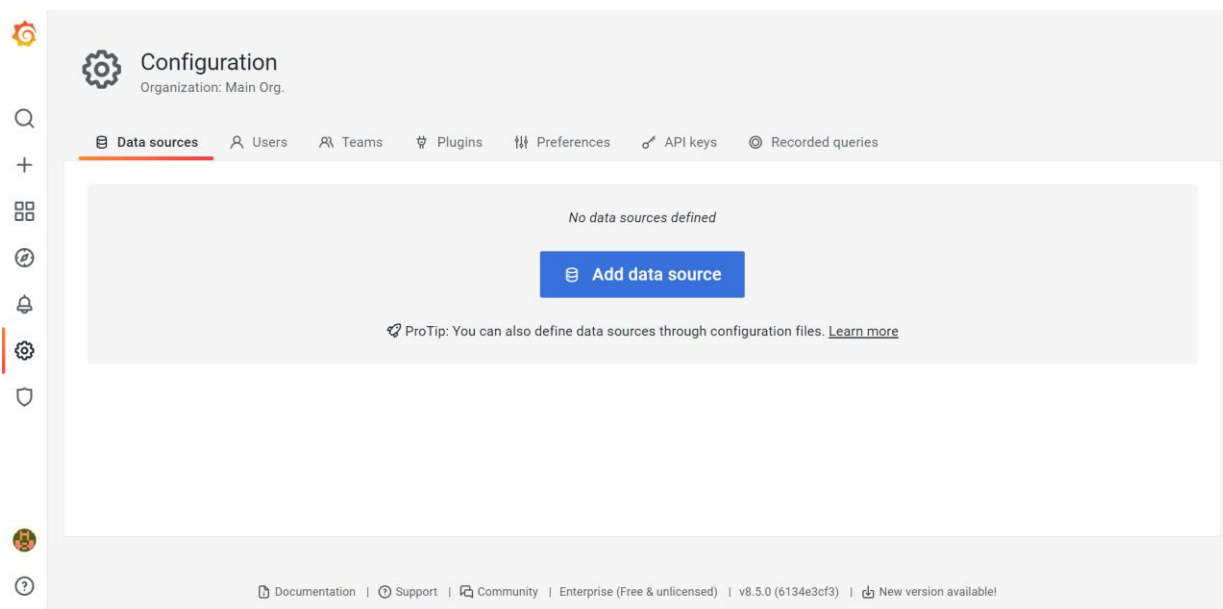
To popularne narzędzie do analizy i wizualizacji danych może zostać połączone z wieloma różnymi źródłami (bazami danych) i przedstawiać pozyskane z nich rekordy i statystyki w postaci różnego rodzaju liczników, wykresów, tabel itd. Dzięki licznym wtyczkom można Grafanę rozszerzyć o dodatkowe źródła danych albo wizualizacje. Na oficjalnej stronie dostępna jest duża liczba gotowych paneli (ang. *dashboard*) stworzonych przez społeczność.

Listing 37 - definicja kontenera z serwerem Grafana napisana w Docker-Compose

```
---
version: '3'
services:
  grafana:
    image: grafana/grafana-enterprise:8.5.0
    container_name: grafana
    ports:
      - "9091:3000"
    volumes:
      - "/opt/masters-configs/monitoring/grafana:/var/lib/grafana"

networks:
  default:
    external:
      name: cluster-network-1
```

Z uwagi na to, że źródłem Grafany może być Prometheus, stos Prometheus-Alertmanager¹⁸-Grafana jest popularnym rozwiązaniem do monitorowania systemów [8]. Metryki są wizualizowane niemal w czasie rzeczywistym (ograniczonym interwałem zapytań Prometheusa), dane przedstawiane na panelach definiując zapytania napisane w języku PromQL.

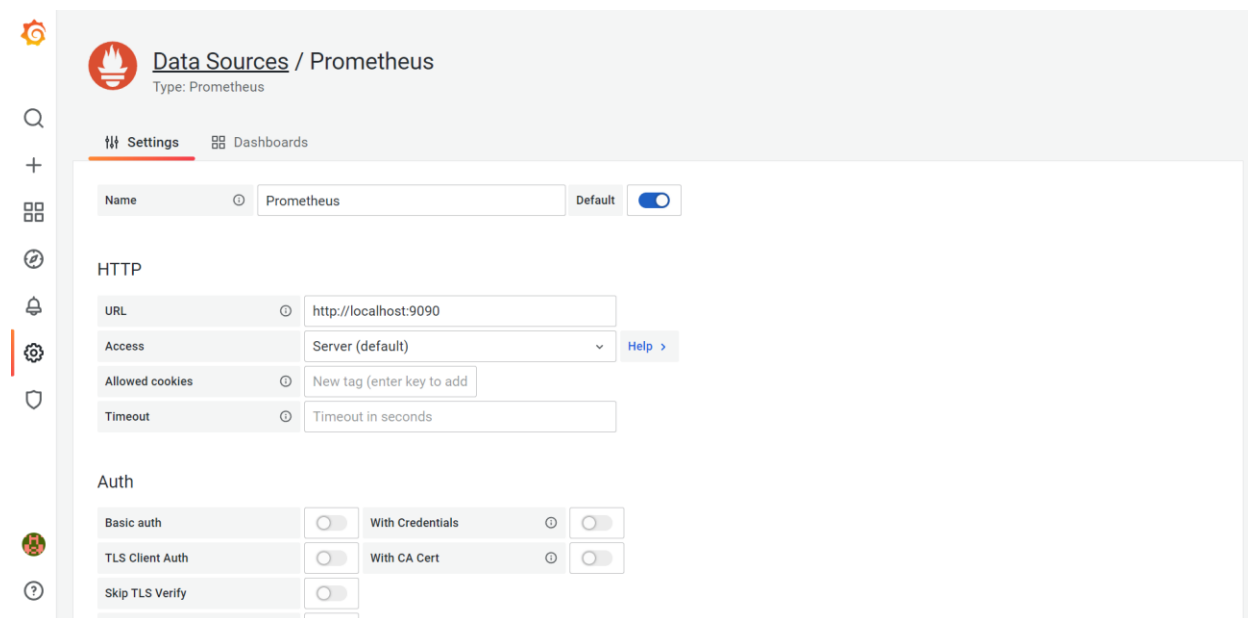


Rysunek 17 - interfejs graficzny Grafany - ekran menu konfiguracji

¹⁸ Alertmanager jest rozwijany wraz z Prometheusem (ale może się też zintegrować z innymi źródłami) narzędziem do zarządzania alarmami. Wysłane do niego powiadomienia mogą zostać pogrupowane, wyciszone na określony czas albo zintegrowane z innymi systemami (jak na przykład komunikatory: MS Teams, Slack) przy użyciu *webhooków*. Dokumentacja Alertmanagera jest dostępna wśród dokumentacji Prometheusa – <https://prometheus.io/docs/alerting/latest/alertmanager> (dostęp 2022-01-09)

Listing 37 przedstawia definicję kontenera z serwerem Grafana – jest ona stosunkowo prosta, ponieważ większość konfiguracji zrealizowano poprzez interfejs graficzny. Wykorzystano oficjalny obraz Grafany Enterprise w wersji 8.5.0 (wydanej w kwietniu 2022). Utworzono wolumen, aby konfiguracja i panele były persystentne.

W ramach konfiguracji zdefiniowano źródło danych (a więc podłączono do serwera Prometheus, Rysunki 17 i 18) oraz utworzono wykresy i panele.



Rysunek 18 - interfejs graficzny Grafany - ekran konfiguracji źródła danych Prometheus

7.3. Eksportery metryk

Większość brokerów oferuje oficjalne eksportery metryk. Jeżeli brakuje oficjalnego – z pomocą przychodzi społeczność. Prometheus jest popularnym frameworkiem i wiele prostych eksporterów można znaleźć na GitHubie tudzież Docker.io. Brokery wiadomości działające na JVM, takie jak Kafka czy ActiveMQ, są kompatybilne z Prometheus JMX Exporter¹⁹.

7.3.1. Kafka

Kafka jest kompatybilna z Prometheus JMX Exporter, ale jego konfiguracja jest dość skomplikowana²⁰. Z pomocą przychodzi gotowe rozwiązanie – danielqsj/kafka_exporter²¹ – rozwijane

¹⁹ https://github.com/prometheus/jmx_exporter (dostęp 2022-12-05) - projekt społeczności, eksporter przeznaczony do działania jako agent uruchomiony obok głównej aplikacji, zbierający i eksportujący dane na temat działania JVM, takie jak zużycie zasobów czy częstotliwość i czas trwania *Garbage Collection*

²⁰ Monitorowanie Kafki przy użyciu JMX Exportera, wraz z jego przykładową konfiguracją, opisano w [1], sekcja „Setting up the Dockerfile, configuring Prometheus.yml, and running the instances”.

²¹ Projekt znajduje się na GitHubie - https://github.com/danielqsj/kafka_exporter (dostęp 2023-01-11). Wszystkie wydane wersje są też publikowane w postaci gotowych obrazów na Docker.io – <https://hub.docker.com/r/danielqsj/kafka-exporter> (dostęp 2023-01-11)

przez społeczność. Eksporter wymaga minimalnej konfiguracji (przedstawionej na Listingu 38), którą realizuje się poprzez parametry uruchomieniowe procesu w kontenerze. Ponieważ w tym przypadku Kafka i Zookeeper pozwalają połączyć się bez autoryzacji, wystarczy tylko podać parametr `kafka.server` z adresem brokera Kafki. Gdyby Kafka działała w modelu rozproszonym, można podać listę adresów brokerów, oddzielonych przecinkami. Można też podać adres Zookeepera w parametrze `zookeeper.server`.

Listing 38 - fragment pliku Docker-Compose z Listingu 1 zawierający definicję kontenera eksportera metryk

```
...
exporter:
  image: danielqsj/kafka-exporter:v1.6.0 #August 2022
  container_name: kafka-exporter
  command: --kafka.server=kafka-broker:29092
  restart: always
  ports:
    - "9001:9308"
...
```

7.3.2. RabbitMQ

Eksporter metryk RabbitMQ uruchomiono instalując wtyczkę *rabbitmq-prometheus* (patrz: 5.2. i Listing 3).

7.3.3. RocketMQ

Istnieje oficjalny eksporter metryk Prometheus dedykowany dla RocketMQ²², ale nie został opublikowany w formie gotowego obrazu dockerowego. Zbudowano go więc własnoręcznie.

Mając już obraz, dodano do pliku Docker-Compose opisanego w 4.3 definicję kontenera eksportera (Listing 39). Zmienne środowiskowe odpowiadają poszczególnym parametrom konfiguracyjnym. Wymagana konfiguracja to:

- numer portu, na którym mają być wystawione metryki,
- wersja brokera RocketMQ,
- adres *nameservera* RocketMQ,
- częstotliwość zbierania statystyk z brokera (format Cron).

Konieczność podania interwału odpytywania serwerów o statystyki może się wydawać niecodzienna. W przeciwieństwie do pozostałych eksporterów, ten działa w sposób asynchroniczny – to znaczy odświeża swoje metryki co określony interwał, a nie za każdym razem, kiedy odpowiada na zapytanie. Metryki przechowuje w swoim *cache'u* i na żądania Prometheusa odpowiada tym, co ma akurat pod ręką.

²² Projekt na Github: <https://github.com/apache/rocketmq-exporter> (dostęp 2023-01-15)

Listing 39 - fragment pliku Docker-Compose - definicja kontenera eksportera RocketMQ

```
exporter:
  image: rocketmq-exporter:latest
  container_name: rocketmq-exporter
  environment:
    SERVER_PORT: 5557
    ROCKETMQ_CONFIG_ROCKETMQ_VERSION: V4_9_3
    ROCKETMQ_CONFIG_NAMESRV_ADDR: rocketmq-nameserver:9876
    TASK_COLLECT_TOPIC_OFFSET: 0/5 * * * * ?
    TASK_COLLECT_CONSUMER_OFFSET: 0/5 * * * * ?
    TASK_COLLECT_BROKER_STATS_TOPIC: 0/5 * * * * ?
  ports:
    - "9004:5557" # metrics
  restart: always
  depends_on:
    - broker
```

7.3.4. ActiveMQ

Nie ma eksportera metryk dedykowanego dla ActiveMQ. Ponieważ ten system kolejkowy napisano w Javie i działa on na JVM, możliwe jest monitorowanie go przy użyciu uniwersalnego agenta Prometheus JMX Exporter (patrz: 7.3.). Plik JAR w wersji 0.17.0 (maj 2022) ściągnięto z oficjalnej strony projektu na GitHub i dodano do definicji obrazu ActiveMQ (Listing 40).

Listing 40 - zawartość pliku Dockerfile, służącego do utworzenia obrazu ActiveMQ wraz z eksporterem metryk

```
FROM openjdk:11.0-jdk-slim
COPY apache-activemq-5.17.1 /opt/apache-activemq-5.17.1
COPY jmx_prometheus_javaagent-0.17.0.jar /opt/jmx_prometheus_javaagent-0.17.0.jar
WORKDIR /opt/apache-activemq-5.17.1/bin
ENTRYPOINT ["/activemq", "console"]
```

Korzystając z takiego obrazu, można uruchomić eksporter metryk, ustawiając odpowiednie parametry uruchomieniowe JVM za pośrednictwem odpowiedniej zmiennej środowiskowej (zazwyczaj JAVA_OPTS, w tym przypadku ACTIVEMQ_OPTS), jak pokazano na Listingu 41. Agent działa na porcie 8080, który zmapowano, umożliwiając połączenie spoza sieci dockerowej.

Listing 41 - zawartość pliku Docker-Compose – definicja kontenera ActiveMQ z eksporterem metryk

```
version: '3'
services:
  activemq:
    image: activemq:5.17.1
    container_name: activemq
    ports:
      - "9005:8080" # jmx-exporter
      - "9100:8161" # console
      - "9101:61616" # data - OpenWire protocol
    environment:
      ACTIVEMQ_OPTS: "-javaagent:/opt/jmx_prometheus_javaagent-0.17.0.jar=8080:/opt/apache-activemq-5.17.1/conf/jmx-config.yml"
    volumes:
      - "/opt/masters-configs/activemq/conf:/opt/apache-activemq-5.17.1/conf"
    restart: always

networks:
  default:
    external:
      name: cluster-network-1
```

7.3.5. NSQ

NSQ nie ma oficjalnego dedykowanego eksportera metryk Prometheus. W Internecie jest kilka małych projektów społeczności, jednak żaden z nich nie jest regularnie rozwijany ani nie działa bez zarzutów²³.

Listing 42 - fragment pliku Docker-Compose - definicja kontenera eksportera metryk NSQ

```
exporter:
  image: harrypiotr/nsq-exporter:1.0
  container_name: nsq-exporter
  environment:
    NSQ_BROKER_SERVER: "nsq-broker:4151"
  depends_on:
    - broker
  ports:
    - "9006:8082"
```

Broker NSQ (nsqd) eksponuje metryki w formacie StatsD, które mają tę przewagę nad Prometheusem, że są bardziej czytelne dla ludzkiego oka. Zdecydowano się na napisanie prostego eksportera, który ściąga metryki StatsD bezpośrednio z brokera, parsuje je i udostępnia je (dzięki wbudowanemu serwerowi HTTP) w postaci kompatybilnej z Prometheusem.

²³ Odnaleziono dwa projekty na Github. [lovoo/nsq_exporter](https://github.com/lovoo/nsq_exporter) (https://github.com/lovoo/nsq_exporter - dostęp 2023-01-14) został nawet wymieniony w oficjalnej dokumentacji Prometheusa. Drugi to, wyglądający na porzucony porzucony, [caozhipan/nsq-prometheus-exporter](https://github.com/caozhipan/nsq-prometheus-exporter) (<https://github.com/caozhipan/nsq-prometheus-exporter> - dostęp 2023-01-14)

Listing 43 - metryki w formacie StatsD zwracane przez punkt końcowy /stats na porcie HTTP brokera. Dla czytelności wycięte zostały nadmiarowe białe znaki

```
nsqd v1.2.1 (built w/go1.16.7)
start_time 2023-01-14T21:49:41Z
uptime 20m1.9041013s

Health: OK

Memory:
heap_objects          11997
heap_idle_bytes       64094208
heap_in_use_bytes     2392064
heap_released_bytes   63963136
gc_pause_usec_100     0
gc_pause_usec_99      0
gc_pause_usec_95      0
next_gc_bytes         4473924
gc_total_runs         0

Topics:
[test-topic-1 ] depth: 0 be-depth: 0 msgs: 9 e2e%:
    [channel-1 ] depth: 9 be-depth: 0 inflt: 0 def: 0 re-q: 0 timeout: 0 msgs: 9 e2e%:

[test-topic-2 ] depth: 3 be-depth: 0 msgs: 3 e2e%:

[test-topic-3 ] depth: 1 be-depth: 0 msgs: 1 e2e%:

Producers: None
```

Listing 44 - metryki zwracane przez napisany eksporter, odpowiadające przedstawionym na Listingu 27

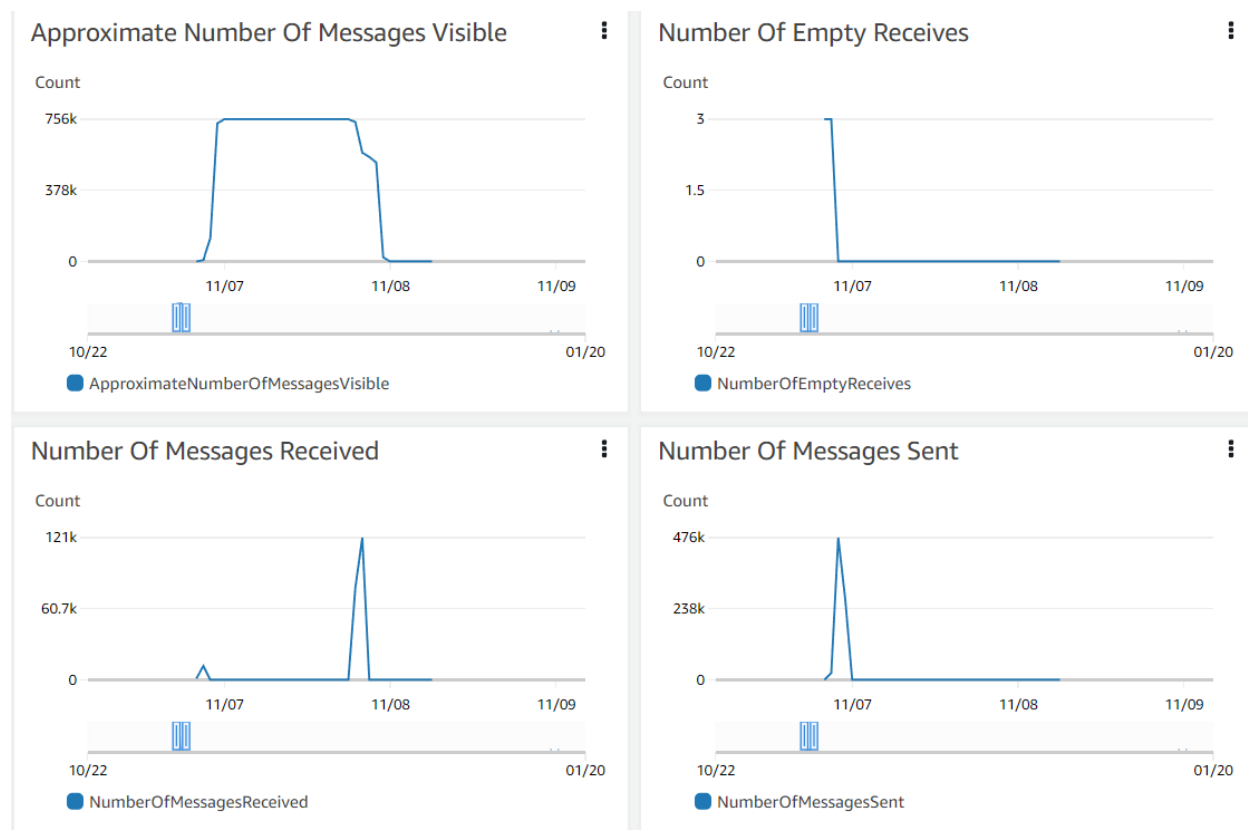
```
nsq_broker_start_time_ms{broker="nsq-broker:4151"} 1.673732981E12
nsq_broker_uptime_ms{broker="nsq-broker:4151"} 1201790.0
nsq_broker_health{broker="nsq-broker:4151"} 1.0
nsq_broker_memory_heap_objects{broker="nsq-broker:4151"} 11997.0
nsq_broker_memory_heap_idle_bytes{broker="nsq-broker:4151"} 6.4094208E7
nsq_broker_memory_heap_in_use_bytes{broker="nsq-broker:4151"} 2392064.0
nsq_broker_memory_heap_released_bytes{broker="nsq-broker:4151"} 6.3963136E7
nsq_broker_memory_next_gc_bytes{broker="nsq-broker:4151"} 4473924.0
nsq_broker_memory_gc_total_runs{broker="nsq-broker:4151"} 0.0
nsq_broker_topic_msgs{topic="test-topic-1", broker="nsq-broker:4151"} 9.0
nsq_broker_topic_depth{topic="test-topic-1", broker="nsq-broker:4151"} 0.0
nsq_broker_topic_bedept{topic="test-topic-1", broker="nsq-broker:4151"} 0.0
nsq_broker_channel_msgs{channel="channel-1", topic="test-topic-1", broker="nsq-broker:4151"} 9.0
nsq_broker_channel_inflt{channel="channel-1", topic="test-topic-1", broker="nsq-broker:4151"} 0.0
nsq_broker_channel_depth{channel="channel-1", topic="test-topic-1", broker="nsq-broker:4151"} 9.0
nsq_broker_channel_def{channel="channel-1", topic="test-topic-1", broker="nsq-broker:4151"} 0.0
nsq_broker_channel_bedept{channel="channel-1", topic="test-topic-1", broker="nsq-broker:4151"} 0.0
nsq_broker_channel_timeout{channel="channel-1", topic="test-topic-1", broker="nsq-broker:4151"} 0.0
nsq_broker_channel_req{channel="channel-1", topic="test-topic-1", broker="nsq-broker:4151"} 0.0
nsq_broker_topic_msgs{topic="test-topic-2", broker="nsq-broker:4151"} 3.0
nsq_broker_topic_depth{topic="test-topic-2", broker="nsq-broker:4151"} 3.0
nsq_broker_topic_bedept{topic="test-topic-2", broker="nsq-broker:4151"} 0.0
nsq_broker_topic_msgs{topic="test-topic-3", broker="nsq-broker:4151"} 1.0
nsq_broker_topic_depth{topic="test-topic-3", broker="nsq-broker:4151"} 1.0
nsq_broker_topic_bedept{topic="test-topic-3", broker="nsq-broker:4151"} 0.0
```

Z powstałego eksportera utworzono obraz dockerowy. Następnie do pliku Docker-Compose dedykowanego dla klastra NSQ dodano definicję kontenera eksportera, przedstawioną na Listingu 42. Eksporter do działania wymaga tylko podania adresu brokera – konfiguruje się go przy użyciu odpowiedniej zmiennej środowiskowej (NSQ_BROKER_SERVER). Na Listingu 43 przedstawiono metryki udostępnione bezpośrednio przez brokera, a na Listingu 44 odpowiadające im metryki Prometheusa.

7.3.6. SQS

Monitoring SQS przy użyciu Prometheusa jest możliwy za pośrednictwem AWS Cloud Watch exportera (opis: [35]) lub prostego eksportera dedykowanego, stworzonego przez społeczność²⁴.

Nie zdecydowano się jednak na instalację żadnego z nich, ponieważ kolejki SQS można monitorować bezpośrednio w konsoli AWS. Poza informacjami takimi jak liczba wiadomości albo liczba subskrybujących klientów, dostępny jest zbiór wykresów wizualizujących działanie kolejki w czasie rzeczywistym. Przykładowy widok panelu monitoringu przedstawiono na Rysunku 19.



Rysunek 19 - fragment ekranu konsoli AWS - wykresy opisujące działanie kolejki SQS

7.3.7. Aplikacje klienckie – *Producer* i *Consumer*

Zarówno generyczny `ProducerController`, jak i klasy rozszerzające, zliczają wysłane wiadomości, aby komponent mógł udostępnić takie liczby w postaci metryk. Zliczanie realizuje metoda `incrementCounter()` (patrz: Listing 9) przyjmująca jako argument nazwę *topica*, tudzież kolejki – powinna być wzywana przy każdym wysłaniu pojedynczej wiadomości. Inna metoda, `getProducerType()`, jest przysyłana przez każdą klasę rozszerzającą i dostarcza nazwę systemu kolejkowego, która jest umieszczana w etykiecie metryki.

²⁴ Projekt na GitHub: <https://github.com/jmal98/sqs-exporter> (dostęp 2023-01-15)

Listing 45 – fragment kodu klasy *ServiceRestController* – metoda *gatherMetrics()*

```
@GetMapping("/metrics")
public void gatherMetrics(HttpServletRequest request, HttpServletResponse response) {

    String responseBody = Arrays
        .stream(context.getBeanDefinitionNames())
        // Odnalezienie wszystkich beanów z nazwą
        // kończącą się na "ProducerController"
        .filter(name -> name.endsWith("ProducerController"))
        .flatMap(name -> {
            List<PrometheusMetric> metrics = new ArrayList<>();
            ProducerController controller =
                (ProducerController) context.getBean(name);
            metrics.add(
                new PrometheusMetric()
                    .withName("producer_messages_sent_total")
                    .withLabel("type",
                        controller.getProducerType())
                    .withValue(controller.getMessagesSentTotal())
            );
            controller
                .getMessagesSent()
                .entrySet()
                .stream()
                .forEach(entry -> metrics.add(
                    new PrometheusMetric()
                        .withName("producer_messages_sent")
                        .withLabel("type",
                            controller.getProducerType())
                        .withLabel("topic",
                            entry.getKey())
                        .withValue(entry.getValue())
                    ));
            return metrics.stream();
        })
        .map(metric -> metric.toString())
        .collect(Collectors.joining("\n"));

    response.setContentType(MediaType.TEXT_PLAIN_VALUE);
    response.setHeader(HttpHeaders.CONTENT_ENCODING, "gzip");
    response.setCharacterEncoding(StandardCharsets.UTF_8.displayName());

    // Prometheus akceptuje tylko enkodowanie GZIP
    try(GZIPOutputStream gzipOutputStream =
        new GZIPOutputStream(response.getOutputStream())) {
        gzipOutputStream.write(responseBody.getBytes(StandardCharsets.UTF_8));
        gzipOutputStream.finish();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

Zbieranie metryk z *beanów* poszczególnych producentów i udostępnianie ich jako metryki zaimplementowano w klasie *ServiceRestController*. Fragment kodu tej klasy, zawierający metodę *gatherMetrics()* przedstawiony został na Listingu 45. Metoda została zmapowana na punkt końcowy „/metrics” aplikacji (przy użyciu springowej adnotacji *@GetMapping*) – domyślny punkt Prometheusa. Kiedy zostaje wezwany, zbierane są liczby wyprodukowanych wiadomości ze wszystkich *beanów* producentów. Następnie metryki są przekształcane do formatu prometheusowego (patrz: Listing 34) i zwracane w odpowiedzi. Co istotne, odpowiedź musi zostać skompresowana w formacie GZIP, o czym pobeżnie informuje [26].

8. Porównanie systemów kolejkowych

W tym rozdziale opisano subiektywne wnioski i obserwacje na temat instalacji i konfiguracji poszczególnych brokerów, implementacji aplikacji klienckich. Wzięto pod uwagę dostępność materiałów, intuicyjność narzędzi, integrację z narzędziami do monitorowania itp. Wykorzystano system monitorowania i aplikacje klienckie do pomiaru przepustowości poszczególnych systemów kolejkowych. Sprawdzone jak niektóre właściwości kolejek wpływają na tę przepustowość, zbadano jak dobrze skaluje się ona wraz ze wzrostem liczby wątków²⁵.

8.1. Dostępność dokumentacji

Kafka, RabbitMQ, NSQ i SQS dysponują obszerną oficjalną dokumentacją. Jeśli chodzi o RocketMQ, to oficjalne materiały są przetłumaczone na język angielski tylko częściowo – na wielu stronach przeważa Hanzi. Tekst po angielsku jest z kolei pełen błędów²⁶. Dokumentacja ActiveMQ zawiera wiele artykułów, ale ma nieintuicyjną strukturę. Niektóre odnośniki odnalezione przez wyszukiwarkę wygasły.

Jeśli chodzi o materiały społeczności – najmniejszą liczbę artykułów można znaleźć dla NSQ i RocketMQ. W przypadku tego drugiego, liczba tekstów w języku chińskim ponownie przeważa nad angielskimi.

8.2. Obraz Docker

RabbitMQ, RocketMQ i NSQ dysponują oficjalnymi, publicznie dostępnymi obrazami dockerowymi. W przypadku RocketMQ można własnoręcznie wygenerować odpowiedni obraz (patrz: 5.3.). Kafka ma kilka dobrze udokumentowanych, regularnie aktualizowanych projektów społeczności. Najgorzej jest w przypadku ActiveMQ – najlepszym rozwiązaniem jest własnoręczne utworzenie obrazu.

8.3. Instalacja i niezbędna konfiguracja

Pod tym względem wygrywa RabbitMQ, który można uruchomić wyłącznie na domyślnej konfiguracji. Podobnie jest z ActiveMQ, o ile definicja zbudowanego obrazu zawiera komendę wejściową.

W przypadku NSQ, należy jedynie określić przeznaczenie (nsqlookupd, nsqd, nsqadmin) danego kontenera, wraz z adresem nsqlookupd (dla nsqd i nsqadmin).

RocketMQ wymaga jeszcze podania adresów rozgłoszeniowych, a Kafka – konfiguracji interfejsów nasłuchujących (ang. *listenerów*). Kafki nie da się też uruchomić w trybie PLAINTEXT, bez autoryzacji, o ile nie doda się do konfiguracji odpowiednich flag.

SQS nie wymaga instalacji, a jedynie podstawowej konfiguracji w konsoli AWS. Utworzenie gotowej do użycia kolejki to kwestia sekund.

²⁵ Dobrze – to znaczy wprost proporcjonalnie do liczby wątków

²⁶ Na przykład: “Before test with tools, we *need* set the nameserver address to system.”, “After finishing *the practice*, we *could* shut down the service[...]” – [31]

8.4. Integracja z narzędziami do monitorowania

Najłatwiej jest zainstalować eksporter metryk RabbitMQ – wystarczy dodać odpowiednią wtyczkę, a broker wystawi wszystkie metryki. Oprócz eksportera przygotowano oficjalny gotowy panel monitoringu dla Grafany²⁷.

Jest kilka eksporterów i *dashboardów* dedykowanych dla Kafki rozwijanych przez społeczność. Można też skorzystać z eksportera JMX.

NSQ jest jedynym systemem kolejkowym, który oferuje wbudowany eksporter metryk – w formacie StatsD. Dla systemów informatycznych, w których stosem technologicznym monitoringu jest na przykład StatsD-Graphite-Grafana, jest to zdecydowanie zaleta.

RocketMQ ma swój dedykowany eksporter, ale nie jest dostępny w formie obrazu dockerowego – trzeba zbudować go własnoręcznie.

Dedykowanych rozwiązań dla ActiveMQ brak. Można skorzystać z uniwersalnego eksportera JMX. Poza tym, jest jeszcze wtyczka do Telegrafa²⁸, która zbiera statystyki odpytując brokera za pośrednictwem HTTP API.

SQS przedstawia wykresy i statystyki w konsoli administracyjnej. Istnieje możliwość zintegrowania go z Prometheusem przy użyciu AWS Cloudwatch (co oznacza poniesienie dodatkowych kosztów) i kolejnego eksportera²⁹.

8.5. Operacje i debugowanie

RabbitMQ (o ile dodano odpowiednią wtyczkę), NSQ, ActiveMQ (wygląda archaicznie) i SQS są dostarczane z wbudowanym interfejsem administratora. Do RocketMQ można doinstalować oficjalny panel administracyjny³⁰ (jest też obraz dockerowy), a do Kafki – skorzystać z jednego z wielu nieoficjalnych rozwiązań (patrz: [42]).

Przydatną funkcjonalnością jest możliwość produkowania/konsumowania wiadomości i zarządzania brokerem za pośrednictwem API HTTP. Oferuje ją, w rozbudowanym, dokładnie udokumentowanym wydaniu, NSQ. ActiveMQ również można sterować za pomocą API [17].

8.6. Rozmiary obrazów Docker

W porównaniu ujęto tylko rozmiary obrazów brokera i opcjonalnych usług zarządzania topologią. Szczegółowe porównanie wykorzystanych obrazów przedstawiono w Tabeli 3. Rysunek 20 przedstawia porównanie sumaryczne każdego z systemów kolejkowych.

²⁷ <https://grafana.com/grafana/dashboards/10991-rabbitmq-overview> (dostęp 2023-01-16)

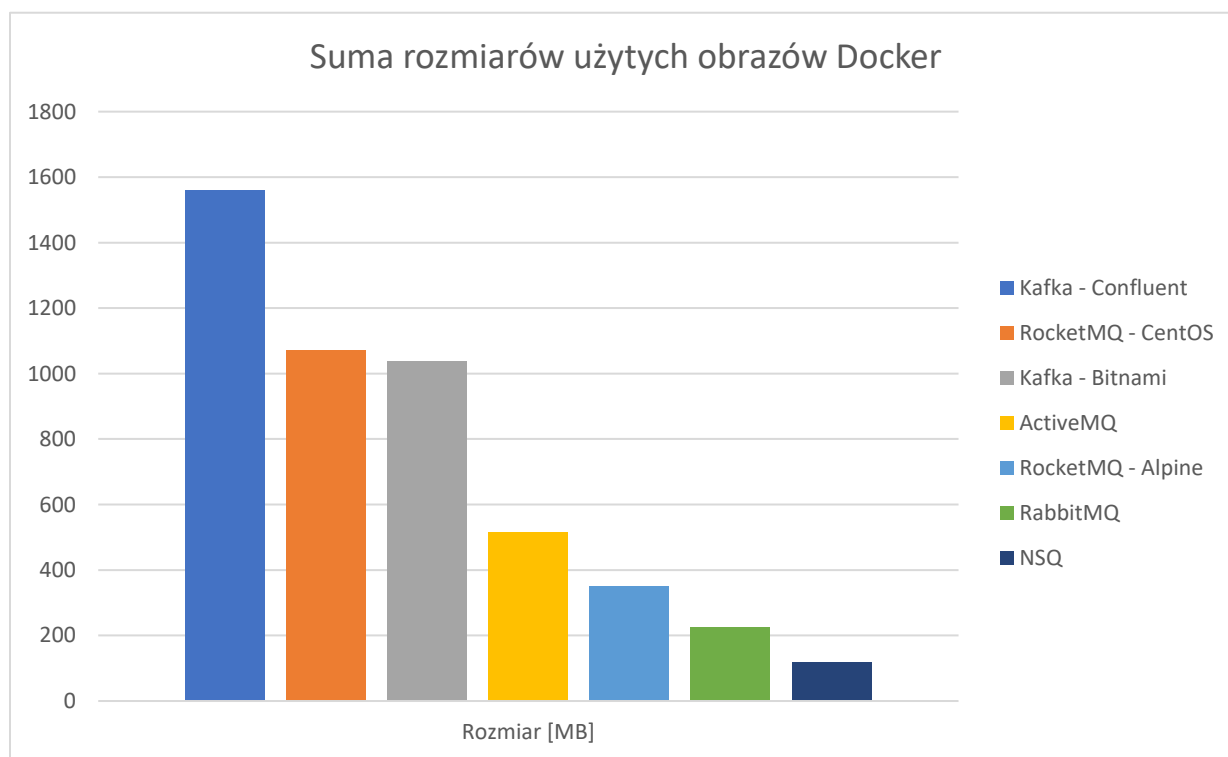
²⁸ Telegraf to oprogramowanie zbierające, przetwarzające i udostępniające metryki, konfigurowalne przy użyciu licznych wtyczek – <https://github.com/influxdata/telegraf> (dostęp 2023-01-16)

²⁹ YACE to narzędzie wystawiające metryki Cloudwatch. Strona projektu na GitHub: <https://github.com/nerdswords/yet-another-cloudwatch-exporter> (dostęp 2023-01-16)

³⁰ Strona projektu na GitHub: <https://github.com/apache/rocketmq-dashboard> (dostęp 2023-01-16)

Tabela 3 - porównanie rozmiarów wykorzystanych obrazów

System kolejkowy	Usługa	Obraz	Rodzaj obrazu	Rozmiar [MB]
Kafka	Zookeeper	bitnami/zookeeper:3.6.4	nieoficjalny - Bitnami	503
	Broker	bitnami/kafka:3.1.2		534
	Zookeeper	confluentinc/cp-kafka:7.0.1	nieoficjalny - Confluent	780
	Broker	confluentinc/cp-zookeeper:7.0.1		780
RabbitMQ	RabbitMQ	rabbitmq:3.9.17	oficjalny	224
RocketMQ	Nameserver	apacherocketmq/rocketmq:4.9.3-alpine	zbudowany przy użyciu generatora	175
	Broker	apacherocketmq/rocketmq:4.9.3-alpine		175
	Nameserver	apacherocketmq/rocketmq:4.9.3-centos	zbudowany przy użyciu generatora	535
	Broker	apacherocketmq/rocketmq:4.9.3-centos		535
ActiveMQ	Broker	activemq:5.17.1	zbudowany własnoręcznie	514
NSQ	NSQLookupD	nsqio/nsq:v1.2.1	oficjalny	59
	Broker	nsqio/nsq:v1.2.1		59



Rysunek 20 - porównanie całkowitego rozmiaru obrazów użytych do instalacji poszczególnych systemów kolejkowych

9. Testy na środowisku

Wszystkie systemy kolejkowe (oprócz SQS – z oczywistych przyczyn), a także aplikacje klienckie i narzędzia do monitorowania zainstalowano w środowisko testowym. Celem jest badanie poprawności instalacji i implementacji, ale także porównywanie między sobą różnych systemów lub różnych konfiguracji tego samego systemu/klienta.

Rolę środowiska testowego pełni maszyna wirtualna, na której działa system operacyjny CentOS 7 64-bit³¹. Specyfikacja maszyny, z uwzględnieniem przydzielonych limitów:

- CPU: Intel Core i5-2400, taktowanie 3.1 GHz, 4 rdzenie,
- RAM: taktowanie 1333 MHz, limit 20 GB,
- Dysk: Crucial MX500, limit 50 GB. Parametry podane przez producenta:
 - sekwencyjny odczyt: 560 MB/s,
 - sekwencyjny zapis: 510 MB/s.

Na maszynie zainstalowano Docker i Docker-Compose. Podczas testów uruchamiano w kontenerach brokery (wraz z usługami pomocniczymi), producentów i/lub konsumentów, a także Prometheusa i Grafanę. Wysyłano wiadomości na kolejki i obserwowano zachowanie systemów na wykresach Grafany.

9.1. Kafka – przepustowość producenta w zależności od ACKS

Acks jest właściwością producenta, określającą, czy powinien poczekać na potwierdzenie od brokera, że wiadomość została na nim zapisana, zanim wyśle następną (patrz: 4.1.). Wydajność dla wyłączonego acks jest średnio trzykrotnie wyższa (patrz: Rysunek 21). Takie ustawienie pozbawia jednak producenta pewności, że wiadomość dotarła do celu. W zależności od przypadku użycia, może jednak być przydatne.

9.2. Kafka – przepustowość producenta w zależności od rozmiaru wiadomości

Zbadano, jak bardzo wielkość przesyłanej wiadomości wpływa na tempo przetwarzania. Doświadczenie przeprowadzono dla wiadomości o rozmiarach od 10 bajtów do 1 megabajta. Okazało się, że wraz z wielkością wiadomości, nieznacznie, zwalnia producent. Wyniki przedstawiono na Rysunku 22.

9.3. Kafka – łączna przepustowość produkcji w zależności od liczby producentów

Sprawdzono, jak liczba nadawców produkujących równocześnie wiadomości na ten sam *topic* wpływa na tempo produkcji. Powtórzono próby dla różnych liczb partycji *topica* – 12 i 50. Z przebiegów pokazanych na Rysunku 23 wynika, że troje producentów nie produkuje razem wiadomości trzy razy szybciej niż jeden – średnia przepustowość wzrosła około 2,6 razy.

³¹ CentOS to darmowa dystrybucja Linux na licencji open-source. Jest często wybieranym systemem operacyjnym serwerów dla przedsiębiorstw (ang. *enterprise*), jako darmowa alternatywa dla RHEL (Red Hat Enterprise Linux), z uwagi na liczne podobieństwa. Obie dystrybucje były rozwijane przez tego samego producenta – Red Hat – do czasu podjęcia decyzji o zakończeniu dalszych prac nad darmowym SO. Źródło: [6].

Co zaskakujące, podniesienie liczby producentów do 6 nie przyniosło zauważalnego skutku w porównaniu do 3. Powodem takiego stanu rzeczy jest konfiguracja Kafki, która domyślnie przeznaczają trzy wątki na obsługę zapytań klientów (patrz: [23], *num.network.threads*).

9.4. Kafka – przepustowość produkcji w zależności od liczby partycji *topica*

Zbadano zależność między liczbą partycji *topica* a średnim tempem produkcji wiadomości. Wyniki przedstawione na Rysunku 24 świadczą, że wzrost liczby partycji nie powoduje spadku wydajności producentów.

9.5. Kafka – przepustowość konsumpcji klientów w jednej i kilku grupach

Sprawdzono, czy troje konsumentów osiągnie większą wydajność, kiedy będą odczytywali wiadomości z tego samego *topica* w ramach jednej grupy, czy trzech różnych. Dla porównania sprawdzono wydajność pojedynczego konsumenta (oczywiście w jednej grupie). Według Rysunku 25:

- pojedynczy konsument przetwarza średnio 70643 wiadomości na sekundę,
- troje konsumentów w tej samej grupie przetwarza łącznie średnio 109661 wiadomości na sekundę (1,55 raza więcej niż pojedynczy konsument),
- troje konsumentów w różnych grupach przetwarza łącznie średnio 131807 wiadomości na sekundę (1,87 raza więcej niż pojedynczy konsument).

Co istotne, wydajność konsumentów znacznie przekracza wydajność producentów (według 9.1. pojedynczy producent produkuje średnio 1421 wiadomości na sekundę, a więc jeden konsument przetwarza wiadomości prawie 50 razy szybciej). Podobną zależność zauważono w przypadku innych systemów kolejkowych, dlatego w dalszych porównaniach skupiono się na producentach. Bardziej prawdopodobnym jest, że to właśnie producent stanie się wąskim gardłem systemu.

9.6. RabbitMQ – przepustowość producenta w zależności od rodzaju kolejki – *durability*

RabbitMQ oferuje trwałe i nietrwałe (w znaczeniu *durability*) kolejki. Porównano szybkość przetwarzania producenta w obu przypadkach. Wyniki przedstawiono na Rysunku 26. Wydajność spadła średnio o 32%.

9.7. RabbitMQ – przepustowość produkcji *point-to-point* a *Pub/Sub*

Zaimplementowane zostały dwa zestawy producentów i konsumentów RabbitMQ – jeden wykorzystujący wzorzec *point-to-point*, a drugi *Pub/Sub*. Na Rysunku 27 porównano wydajność obu producentów. Wykorzystywane kolejki były typu *durable*, a wszystkie wiadomości miały po 100 bajtów.

9.8. RocketMQ – przepustowość producenta w zależności od rozmiaru wiadomości

Wyniki przedstawiono na Rysunku 28. W przeciwieństwie do Kafki, wielkość nie wpłynęła zauważalnie na liczbę przesłanych wiadomości na sekundę.

9.9. RocketMQ – łączna przepustowość produkcji w zależności od liczby producentów

Podobnie jak w 9.3., zbadano sumaryczne tempo przetwarzania dla 1, 3, 6, 12 i 24 równolegle pracujących producentów. Wyniki ilustruje Rysunek 29. Średnia liczba wszystkich produkowanych wiadomości na sekundę to:

- 1251 dla 1 producenta,
- 2420 dla 3 producentów (1,93 razy więcej niż 1 producent),
- 3727 dla 6 producentów (2,98 razy więcej niż 1 producent),
- 5625 dla 12 producentów (4,50 razy więcej niż 1 producent),
- 7301 dla 24 producentów (5,84 razy więcej niż 1 producent).

9.10. ActiveMQ – przepustowość produkcji *point-to-point* a *Pub/Sub*

Analogicznie do 9.7., porównano przepustowość producentów w zależności od tego, wykorzystują kolejkę, czy *topic*. Wszystkie wiadomości miały wielkość 100 bajtów. Producenci pracowali w trybie persystentnego dostarczenia wiadomości. W przypadku *topicu* utworzono wcześniej konsumenta typu *DurableSubscriber*. Gdyby tego nie zrobiono, przepustowość byłaby większa, ponieważ broker, w obliczu braku konsumentów, którzy mieliby odczytać wiadomość, nie zapisywałby żadnej z nich. Wyniki przedstawione na Rysunku 30 świadczą, że zastosowanie kolejki albo *topica* nie wpływa na wydajność producenta, która jest niska w porównaniu do innych brokerów.

9.11. NSQ – przepustowość produkcji buforowanej i niebuforowanej

Wykorzystane biblioteki *nsq-j* (patrz: 6.5.) umożliwiają wysyłanie wiadomości w trybie standardowym i buforowanym. Nie jest to cecha samego systemu kolejkowego, a aplikacji klienckiej. Jak pokazano na Rysunku 31, zastosowanie bufora spowodowało znaczny, średnio ponad siedemnastokrotny, wzrost wydajności.

9.12. SQS – łączna przepustowość produkcji w zależności od liczby producentów

Zbadano, jak dobrze skaluje się łączne tempo produkcji wiadomości dla 1, 3, 6, 12 i 24 producentów. Wiadomości miały rozmiar 100 B. Z wykresu na Rysunku 32 wynika, że średnia liczba produkowanych wiadomości na sekundę to:

- 21,3 dla 1 producenta,
- 65,1 dla 3 producentów (3,06 razy więcej niż 1 producent),
- 130 dla 6 producentów (6,10 razy więcej niż 1 producent),
- 259 dla 12 producentów (12,16 razy więcej niż 1 producent),
- 522 dla 24 producentów (24,51 razy więcej niż 1 producent).

Pojedynczy producent ma niską wydajność w porównaniu do innych systemów kolejkowych. Rośnie ona jednak niemal wprost proporcjonalnie do liczby klientów. Wyniki potwierdzają zapewnienia producenta w [9], że przepustowość wzrasta liniowo wraz z liczbą klientów.

9.13. SQS – łączna przepustowość konsumpcji w zależności od liczby konsumentów

Podobne doświadczenie jak w 8.13. wykonano dla 1, 3, 6 i 12 konsumentów. Wszystkie wiadomości miały rozmiar 100 B. Wyniki przedstawiono na wykresie na Rysunku 33, wynika z nich, że średnia łączna konsumpcja (wyrażona w liczbie wiadomości na sekundę) była równa:

- 11,2 dla 1 konsumenta,
- 33,9 dla 3 konsumentów (3,03 razy więcej niż 1 konsument),
- 66,4 dla 6 konsumentów (5,93 razy więcej niż 1 konsument),
- 133 dla 12 konsumentów (11,88 razy więcej niż 1 konsument).

A więc podobnie jak w przypadku produkcji, konsumpcja rosła niemal liniowo wraz z liczbą klientów.

SQS jest jedynym spośród badanych systemów kolejkowych, dla którego wydajność konsumpcji jest niższa niż wydajność produkcji. Może to być spowodowane tym, że aplikacja kliencka jako jedyna odbiera wiadomości z kolejki metodą *Pull* – w przypadku SQS nie ma innej możliwości.

9.14. Porównanie wyników

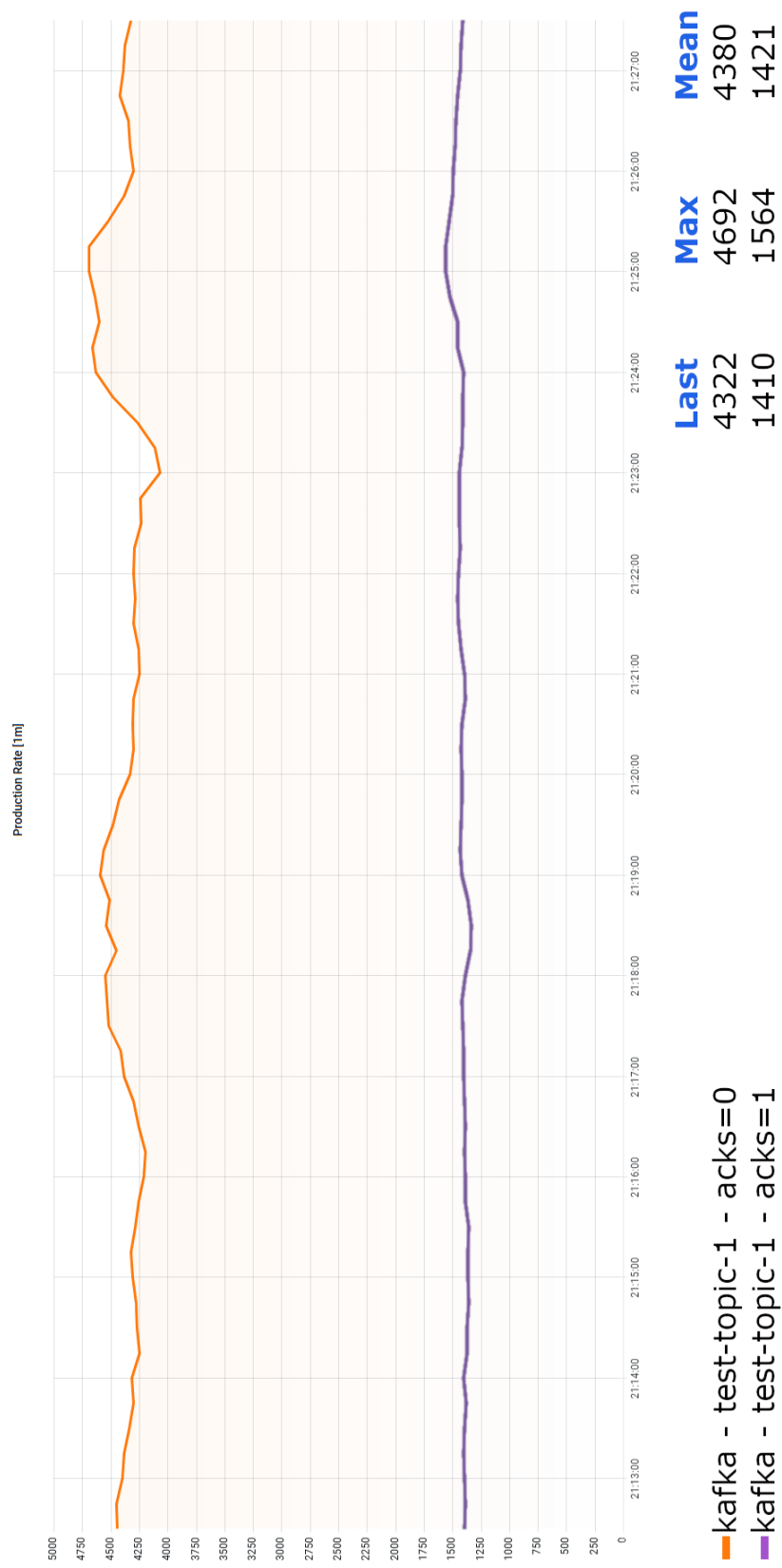
Porównanie przepustowości produkcji i konsumpcji poszczególnych systemów kolejkowych przedstawiono w Tabeli 4. Każdy spośród brokerów (nawet SQS, jeżeli przyjąć nieograniczoną skalowalność zgodnie z zapewnieniami producenta) umożliwia produkcję i konsumpcję z przepustowością rzędu dziesiątek tysięcy wiadomości na sekundę (dla pojedynczego *tematu*, kolejki). Jest to stosunkowo duża wydajność, wystarczająca dla większości przypadków użycia.

Tabela 4 – zestawienie przepustowości poszczególnych systemów kolejkowych

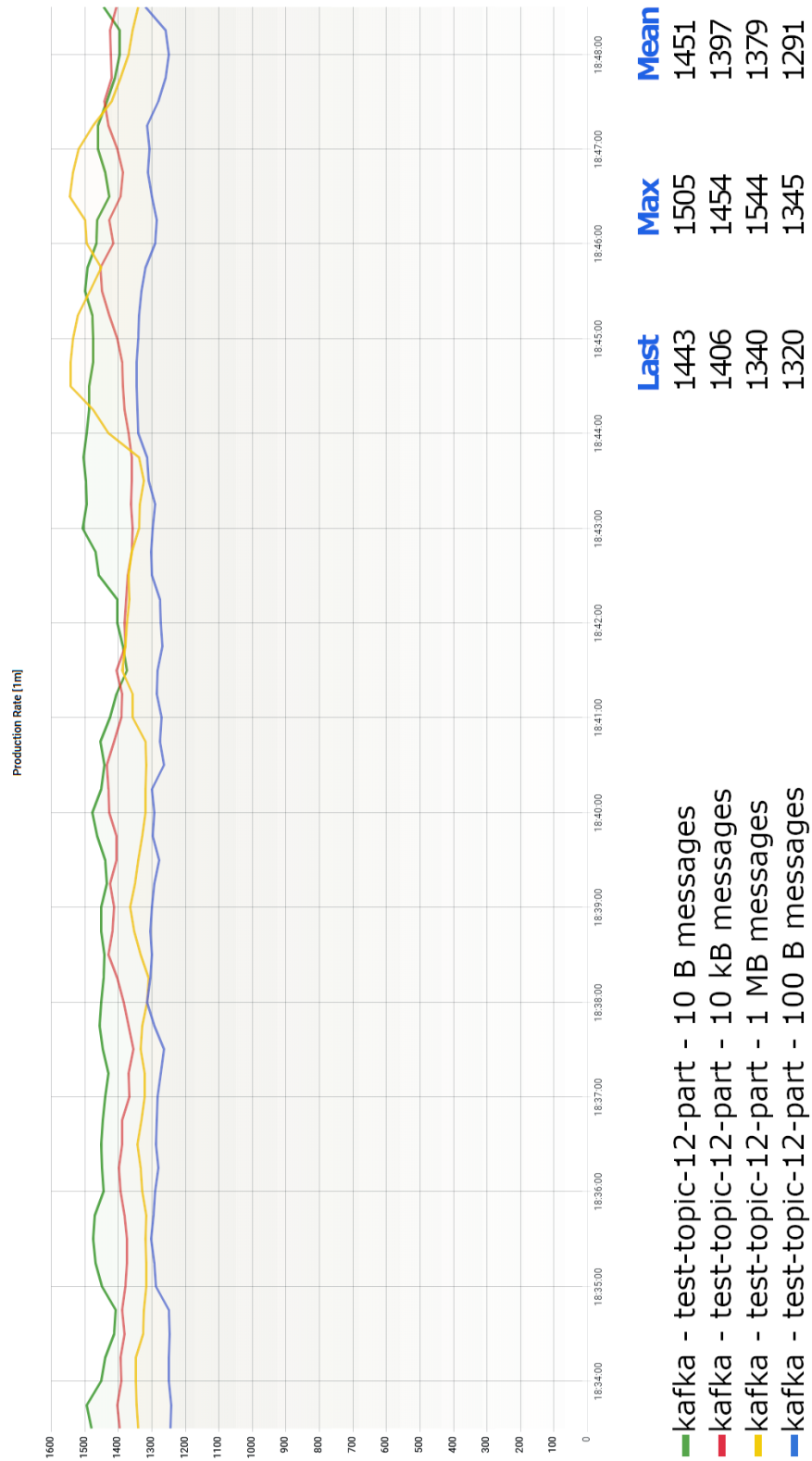
System kolejkowy	Średnia przepustowość produkcji ³² [wiadomości wysłane/s]	Średnia przepustowość konsumpcji ³³ [wiadomości odebrane/s]
Kafka	1 421	70 643
RabbitMQ – <i>point-to-point</i>	15 226	42 934
RabbitMQ – <i>Pub/Sub</i>	5 426	22 378
RocketMQ	1 251	54 124
ActiveMQ – <i>point-to-point</i>	548	7 042
ActiveMQ – <i>Pub/Sub</i>	555	8 300
NSQ	2 911	53 482
SQS	21	11

³² Dla 1 producenta i wiadomości wielkości 100 B. Dla RabbitMQ i ActiveMQ kolejka trwała (*durable, persistent delivery mode*).

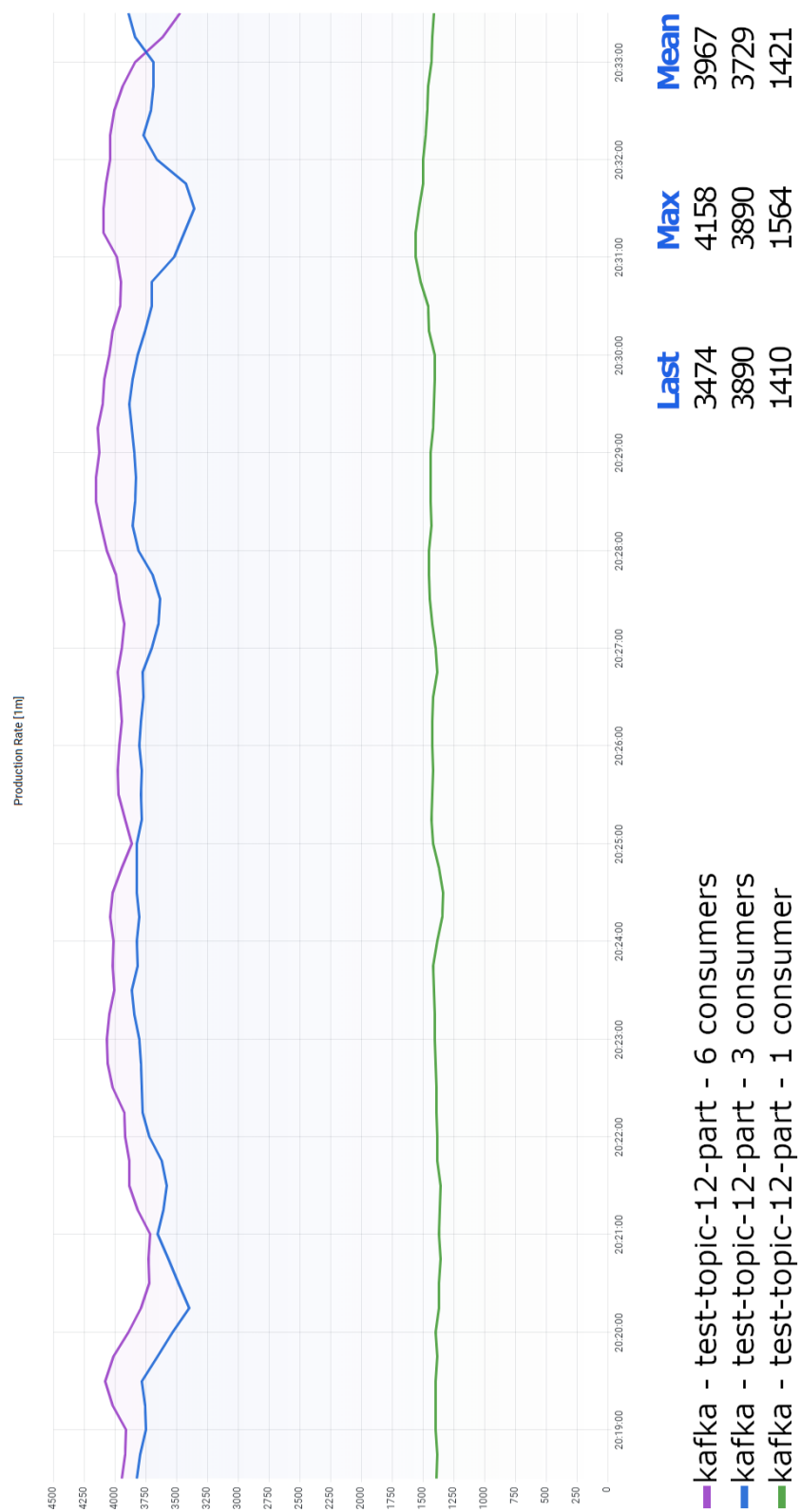
³³ Dla 1 konsumenta i wiadomości wielkości 100 B.



Rysunek 21 - porównanie przepustowości producentów Kafki dla acks równego 0 i 1



Rysunek 22 - porównanie przepustowości producentów Kafki dla różnych rozmiarów przesyłanych wiadomości, acks = 1



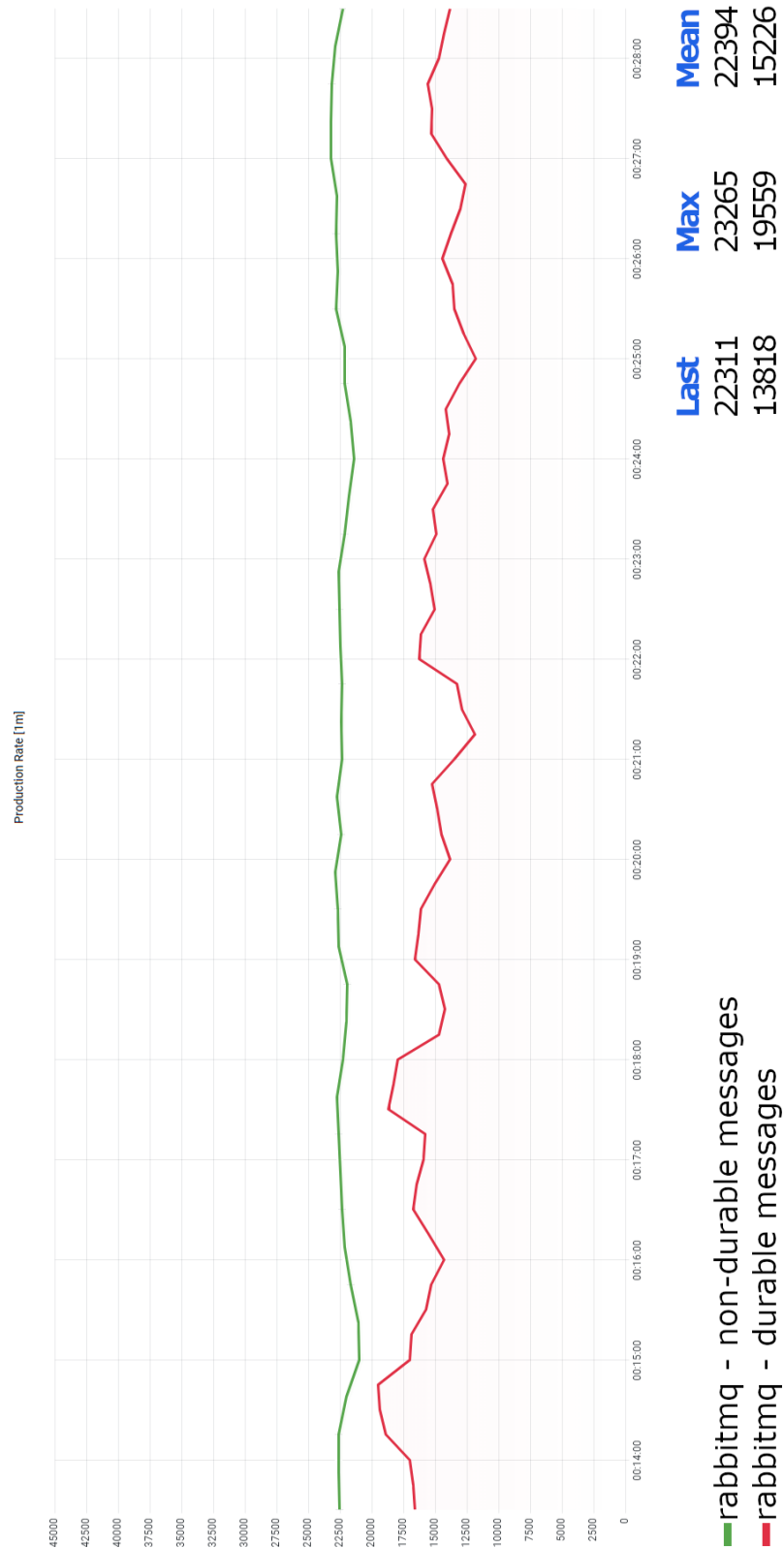
Rysunek 23 - porównanie łącznej przepustowości produkcji Kafki dla różnej liczby producentów równoległe produkujących wiadomości na ten sam topic, acks = 1



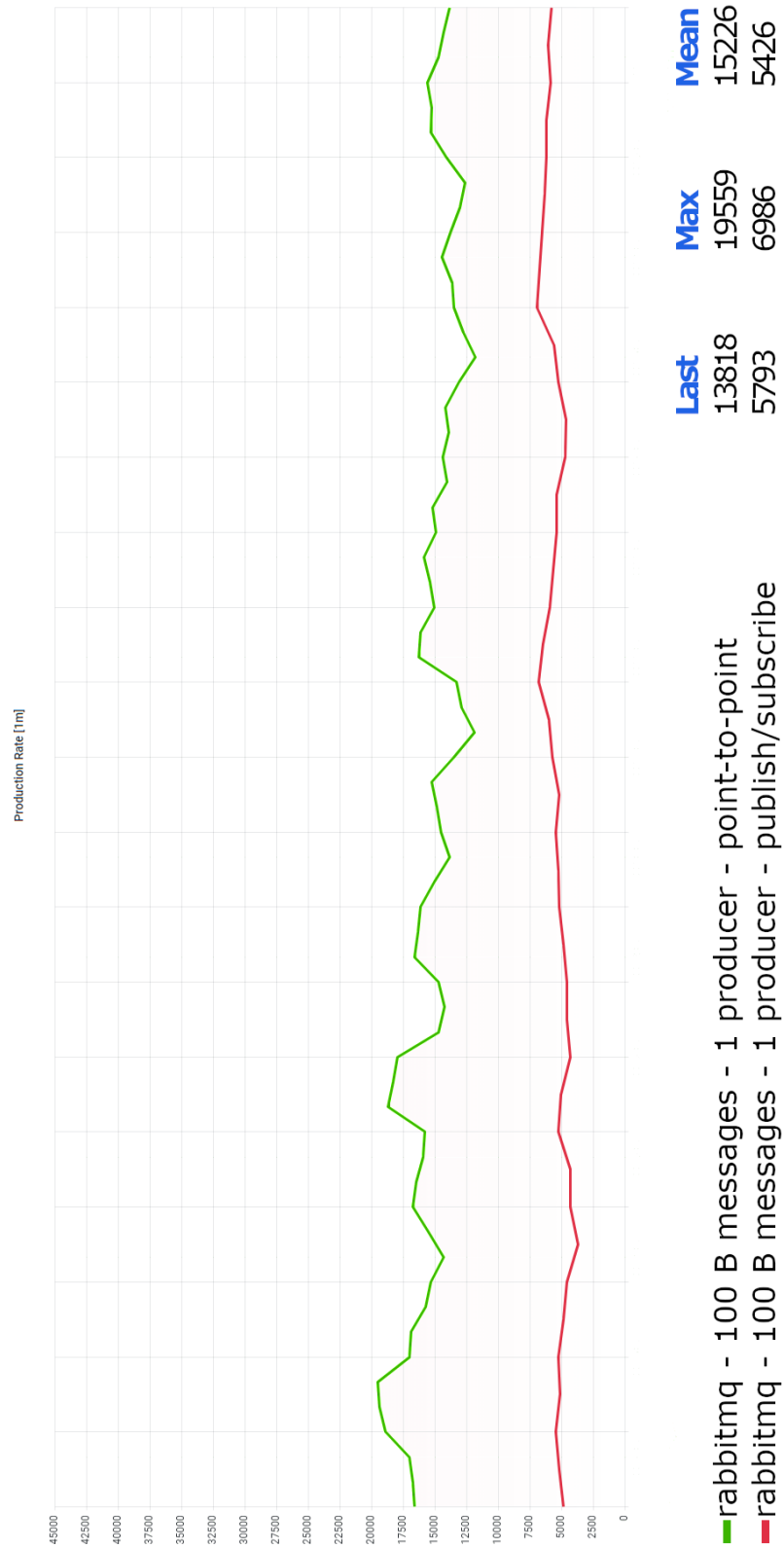
Rysunek 24 - porównanie przepustowości produkcji Kafki dla różnej liczby partycji topica



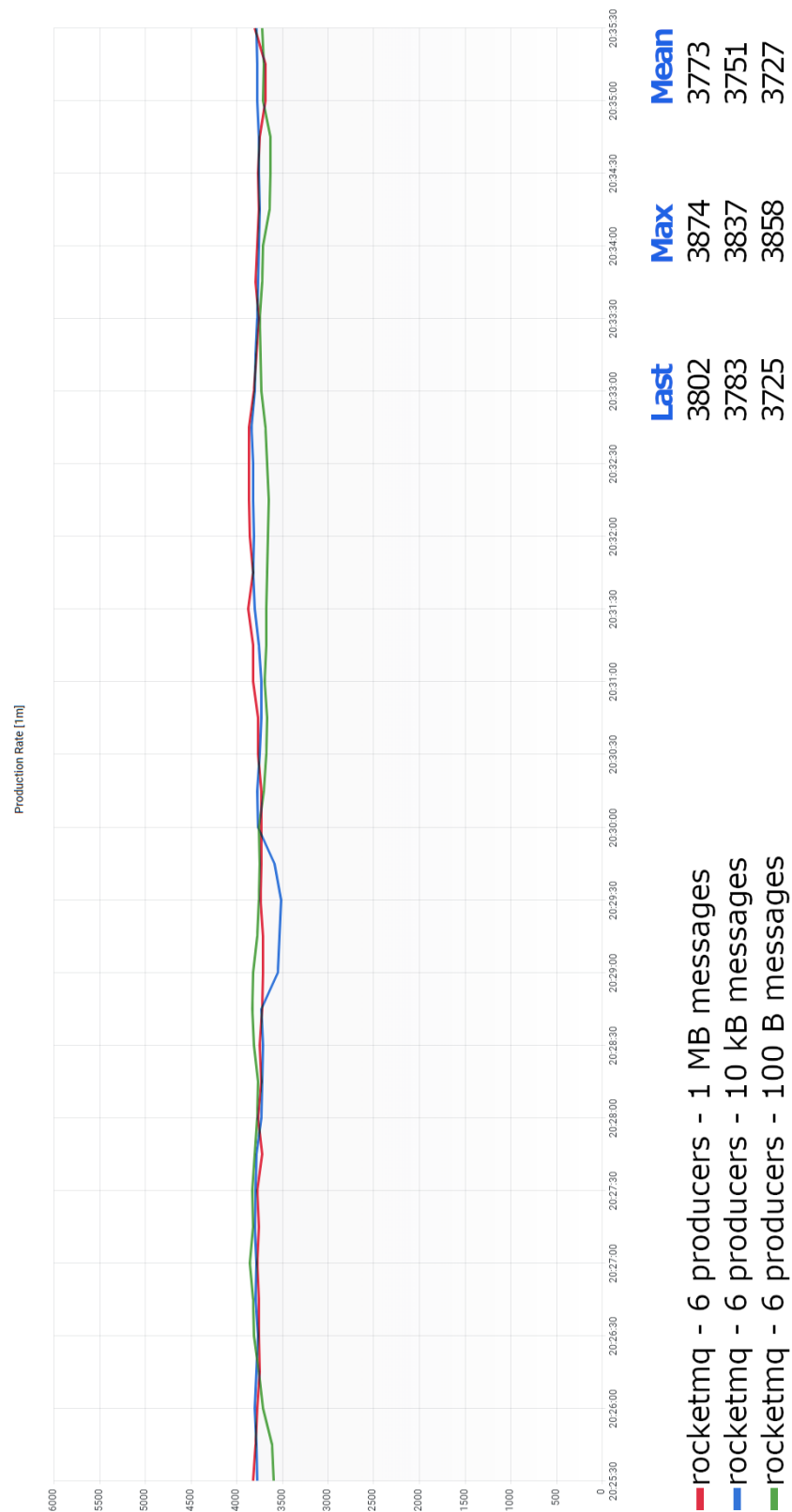
Rysunek 25 - porównanie przepustowości konsumentów Kafki w tej samej i w różnych grupach, $acks = 1$



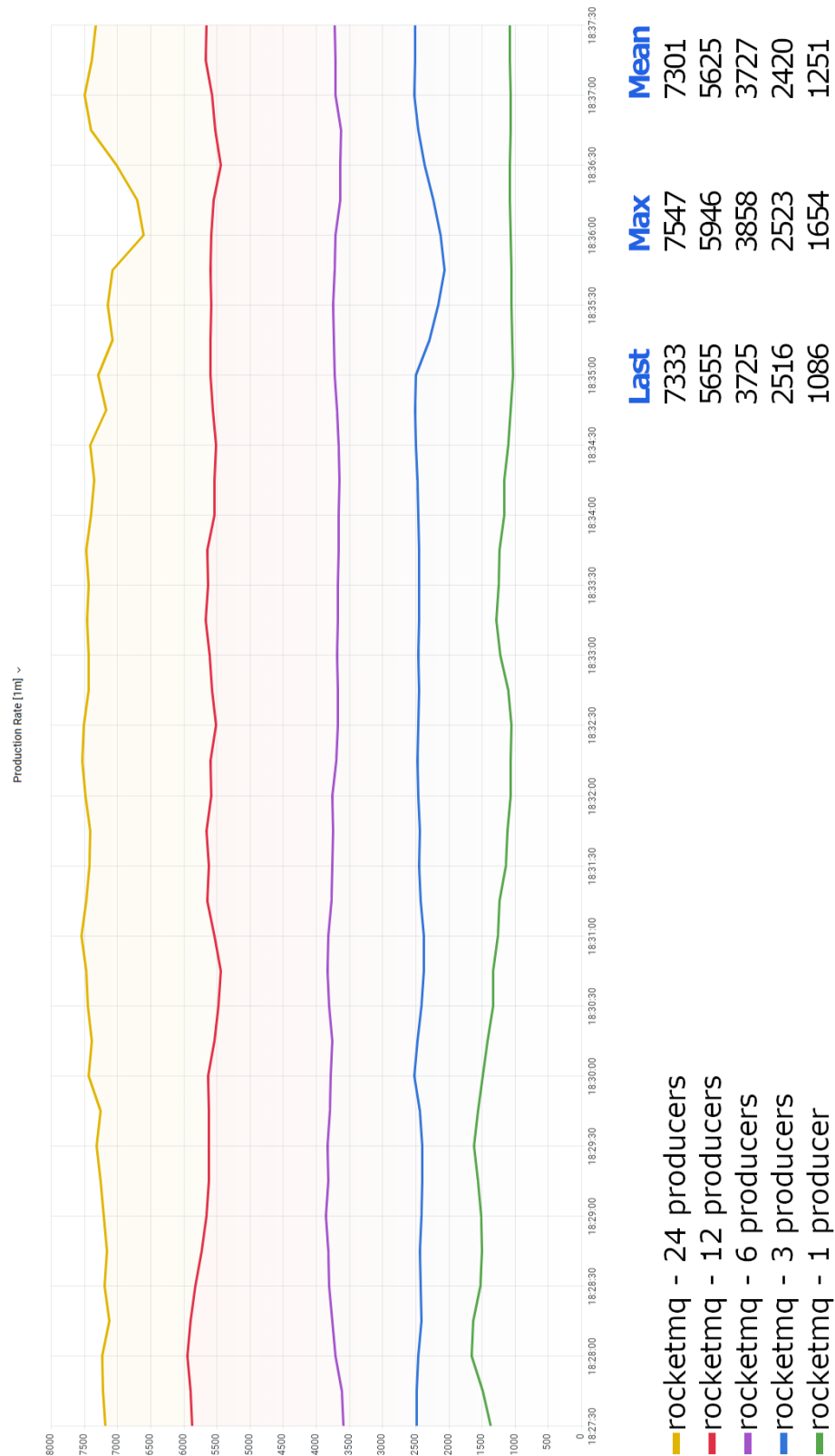
Rysunek 26 - porównanie przepustowości producenta RabbitMQ dla kolejek durable i non-durable



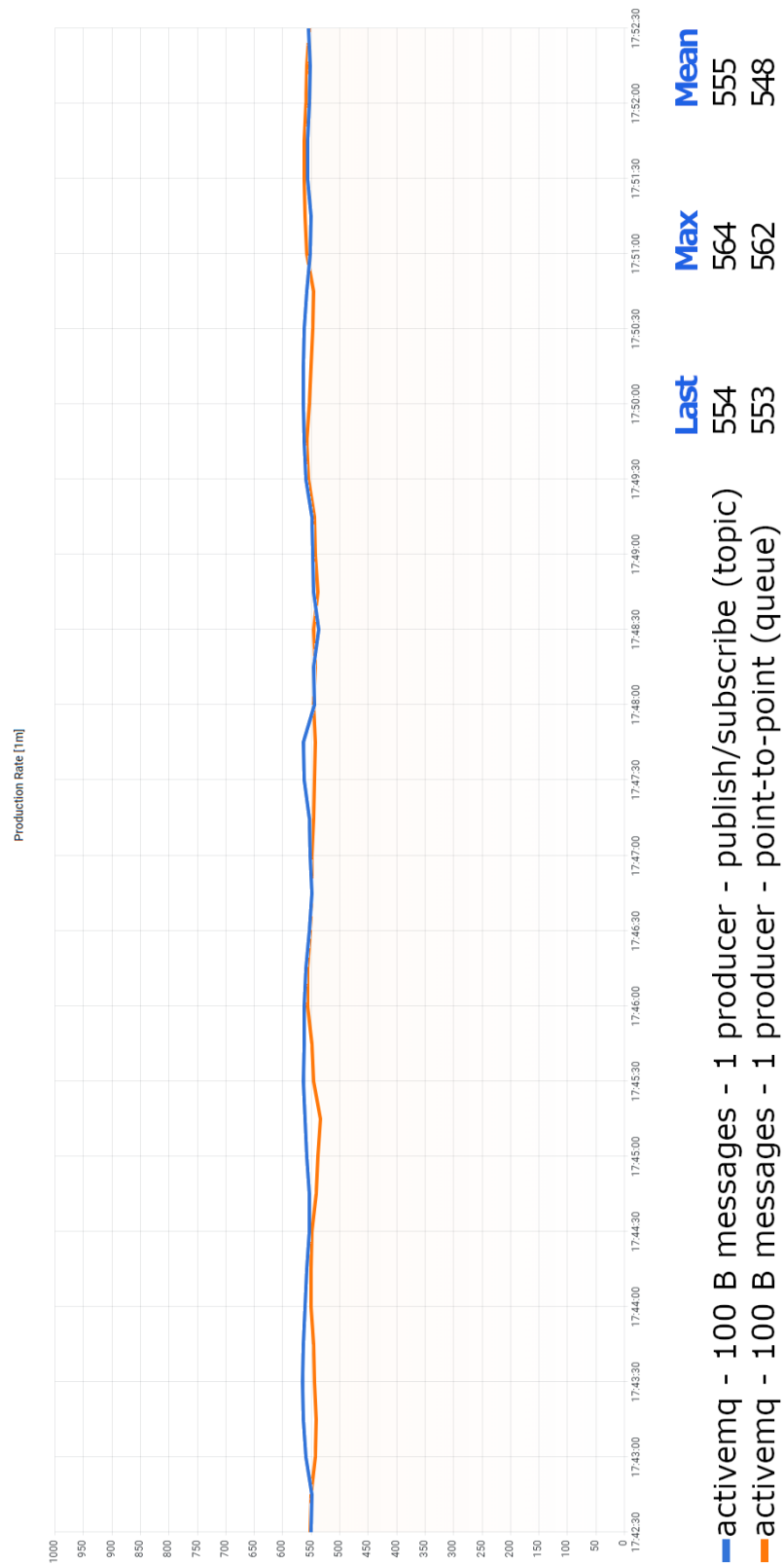
Rysunek 27 – porównanie przepustowości producentów dla implementacji point-to-point i Pub/Sub



Rysunek 28 - porównanie przepustowości producenta RocketMQ w zależności od rozmiaru przesyłanych wiadomości



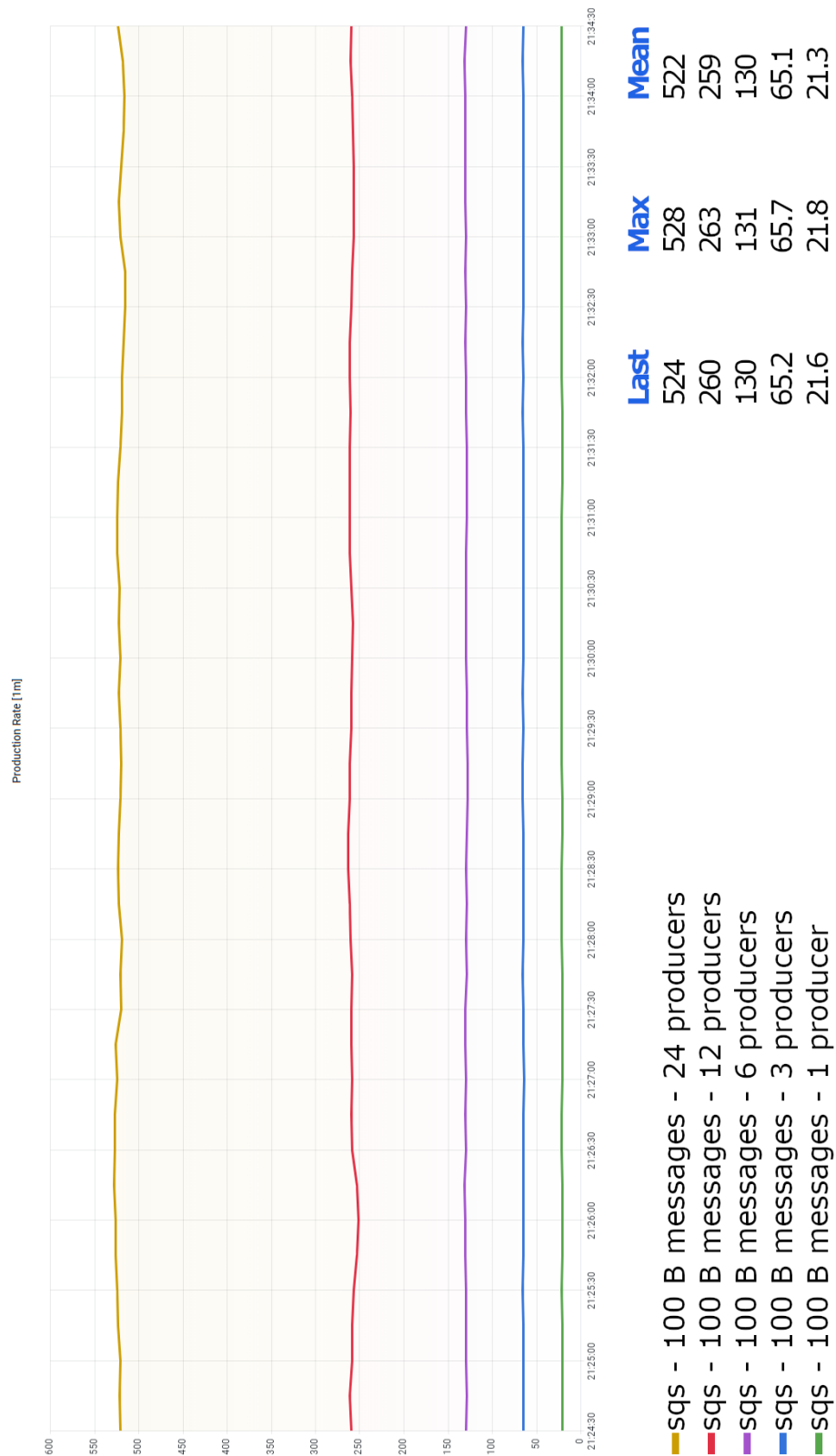
Rysunek 29 - porównanie łącznej przepustowości produkcji RocketMQ dla różnej liczby producentów równoległe produkujących wiadomości na ten sam topic



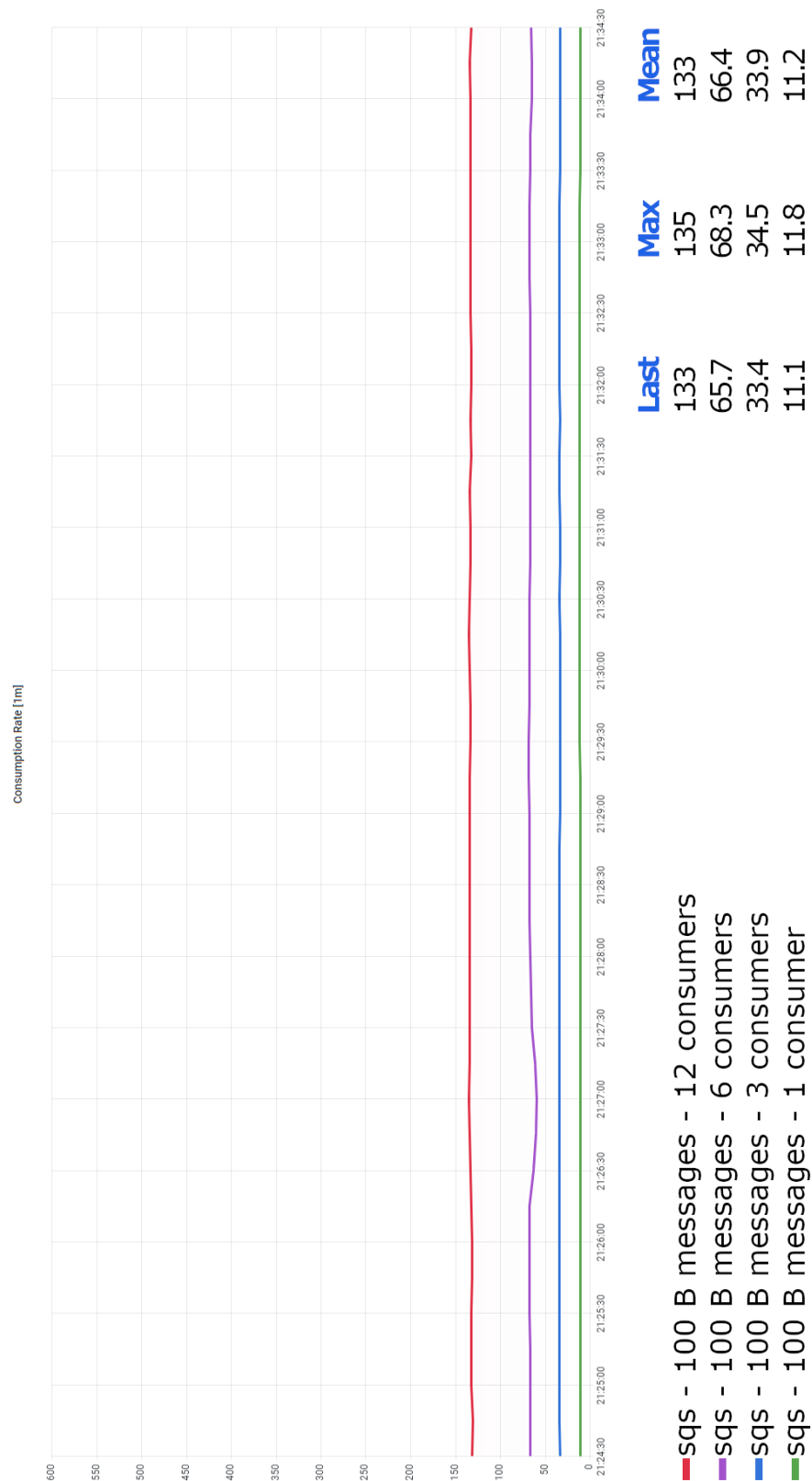
Rysunek 30 – porównanie przepustowości producentów dla implementacji point-to-point (z wykorzystaniem kolejki, ang. queue) i Pub/Sub (z wykorzystaniem topica)



Rysunek 31 - porównanie wydajności producenta NSQ wysyłającego wiadomości w trybie buforowanym i niebuforowanym



Rysunek 32 - porównanie łącznej przepustowości produkcji SQS dla różnej liczby producentów równoległe produkujących wiadomości na tę samą kolejkę



Rysunek 33 - porównanie łącznej przepustowości konsumpcji SQS dla różnej liczby konsumentów równoległe konsumujących wiadomości z tej samej kolejki

10. Podsumowanie

Niniejszy rozdział przedstawia wnioski na podstawie doświadczeń z porównywanymi brokerami wiadomości.

Model hybrydowy, na przykładzie Kafki albo RocketMQ jest uniwersalny. Może zastąpić kolejki działające we wzorcu *point-to-point* lub *Pub/Sub*, w zależności od tego, jak zorganizowano konsumentów i grupy. Istotną cechą jest podział *topiców* na partycje (albo kolejki – w przypadku RocketMQ). *Topic* z jedną partycją zapewnia odbiór komunikatów w kolejności FIFO. W zależności od konfiguracji *topiców* i grup, jeden serwer (lub właściwie klaster) może mieć kilka różnych zastosowań.

Kafka jest najbardziej zaawansowanym, współczesnym brokerem wiadomości. Jest stale rozwijana i cieszy się dużą popularnością, co owocuje integracjami z innymi popularnymi systemami, artykułami, poradnikami albo narzędziami wytworzonymi przez społeczność. Oferuje kilka ciekawych pomysłów, jak na przykład Kafka Streams, które umożliwiają implementację na wiele różnych sposobów. Kafka wymaga dużego nakładu pracy włożonego w konfigurację, a niektóre domyślne ustawienia mogą wydać się mylące (patrz: 9.8.).

RabbitMQ jest kolejką dość lekką (ang. *lightweight*). Ma rozbudowaną dokumentację i jest umiarkowanie popularna. Zaletą są liczne wtyczki, rozszerzające serwer o dodatkowe funkcje. RabbitMQ sprawia wrażenie narzędzia zbudowanego jako prosta implementacja wzorca *point-to-point*, które następnie niezgrabnie rozwijano. Tryb *Pub/Sub* w rzeczywistości oparty jest o kolejki *point-to-point*. Podstawowe biblioteki klienckie domyślnie korzystają z nietrwałych kolejek (w znaczeniu *durability*).

RocketMQ jest klonem Kafki³⁴ oferującym kilka ciekawych, oryginalnych rozwiązań (patrz: 4.4.). Niestety, brakuje mu dokumentacji, nawet tej oficjalnej, nie mówiąc o dodatkowych materiałach, w języku angielskim. Nie cieszy się popularnością zachodniej społeczności i trudno go polecić, ze względu na brak wsparcia.

ActiveMQ jest przestarzały. Widać to np. po archaicznym wbudowanym panelu administracyjnym brokera, braku wsparcia dla Dockera, czy przesadnie skomplikowanych w użyciu oficjalnych bibliotekach klienckich (patrz: 6.4.). System kolejkowy nadal jest rozwijany, wraz z równoległym projektem, ActiveMQ Artemis. Artemis ma, po osiągnięciu dojrzałości, zastąpić klasyczne ActiveMQ, zaczynając od wersji 6.0 – jak informują twórcy w [16]. Być może wniesie to powiew świeżości do projektu.

NSQ to prosta i lekka (najlżejsza z porównywanych) kolejka wiadomości. Ma kilka zalet, takich jak intuicyjna dokumentacja, ładny wbudowany panel administracyjny, udostępnienie HTTP API, czy proste w użyciu biblioteki klienckie (nieoficjalne). Jest implementacją modelu hybrydowego, podobnego do Kafki i RocketMQ, ale zubożonego o podział *topiców* na partycje (lub kolejki) – to wada, gdyż ten mechanizm wzbogaca system o dodatkowe przypadki użycia. Projekt NSQ najwyraźniej został porzucony – ostatnia wydana wersja jest datowana na sierpień 2021.

SQS to w niektórych przypadkach użycia dobra alternatywa dla własnoręcznej instalacji brokera. Fakt, że jest rozwiązaniem płatnym, nie stawia go na przegranej pozycji – utrzymanie infrastruktury, na której miałyby być zainstalowany darmowy system kolejkowy, również kosztuje. Zaletami SQS jest wysoka skalowalność, duża popularność i wsparcie zapewniane przez Amazona. Istotnym jest, że koszty są proporcjonalne do liczby przesyłanych wiadomości, a w mniejszym stopniu do ich rozmiaru, albo liczby kolejek. W projekcie, w którym liczba przesyłanych komunikatów jest niewielka, a istotna jest niezawodność, SQS jest dobrym wyborem. Nic nie stoi też na przeszkodzie, by używać go komplementarnie z innymi brokerami.

³⁴ Jak piszą sami twórcy – inżynierowie Alibaba – w [33], początkowo korzystali z Kafki. Kiedy okazało się, że nie do końca spełnia ich wymagania, stworzyli własne rozwiązanie.

Literatura

- [1] Ahuja, Madhur. 2022. *Kafka Monitoring Using Prometheus* – <https://www.metricfire.com/blog/kafka-monitoring> (dostęp 2023-01-11)
- [2] Beznos, Marta. 2023. *Microservices vs monolith: Which architecture is the best choice for your business?* – <https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business> (dostęp 2023-01-10)
- [3] Blog Baeldung. 2020. *Apache RocketMQ with Spring Boot* – <https://www.baeldung.com/apache-rocketmq-spring-boot> (dostęp 2023-01-15)
- [4] Blog Bitly. *NSQ: realtime distributed message processing at scale* – <https://word.bitly.com/post/33232969144/nsq> (dostęp 2023-01-09)
- [5] Blog Fellow.app. *Synchronous vs. Asynchronous Communication: A Guide For Managers* – <https://fellow.app/blog/remote/synchronous-vs-asynchronous-communication-guide-for-managers> (dostęp 2023-01-01)
- [6] Blog Red Hat. *What is CentOS?* – <https://www.redhat.com/en/topics/linux/what-is-centos> (dostęp 2023-01-15)
- [7] Dhanushka, Dunith. 2020. *The Stuff That Every Developer Should Know About Message Queues* – <https://medium.com/event-driven-utopia/the-stuff-that-every-developer-should-know-about-message-queues-a9452ac9c9d> (dostęp 2023-01-01)
- [8] Dokumentacja AWS. *Amazon SQS Features* – <https://aws.amazon.com/sqs/features> (dostęp 2022-01-08)
- [9] Dokumentacja AWS. *Increasing throughput using horizontal scaling and action batching* – <https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-throughput-horizontal-scaling-and-batching.html> (dostęp 2023-01-17)
- [10] Dokumentacja AWS. *What is Pub/Sub Messaging?* – <https://aws.amazon.com/pub-sub-messaging> (dostęp 2023-01-01)
- [11] Dokumentacja AWS. *What is a Message Queue?* – <https://aws.amazon.com/message-queue> (dostęp 2023-01-01)
- [12] Dokumentacja ActiveMQ. *Getting Started* – <https://activemq.apache.org/getting-started> (dostęp 2023-01-12)
- [13] Dokumentacja ActiveMQ. *Hello World* – <https://activemq.apache.org/hello-world> (dostęp 2023-01-15)
- [14] Dokumentacja ActiveMQ. *How do I make messages durable* – <https://activemq.apache.org/how-do-i-make-messages-durable> (dostęp 2023-01-15)
- [15] Dokumentacja ActiveMQ. *How does a Queue compare to a Topic* – <https://activemq.apache.org/how-does-a-queue-compare-to-a-topic> (dostęp 2023-01-17)
- [16] Dokumentacja ActiveMQ. *New features in 6.0* – <https://activemq.apache.org/new-features-in-60> (dostęp 2023-01-17)
- [17] Dokumentacja ActiveMQ. *REST* – <https://activemq.apache.org/rest> (dostęp 2023-01-16)
- [18] Dokumentacja ActiveMQ. *What is the difference between persistent and non-persistent delivery?* – <https://activemq.apache.org/what-is-the-difference-between-persistent-and-non-persistent-delivery> (dostęp 2023-01-15)
- [19] Dokumentacja DataStax. *How Apache Kafka messages are written* – <https://docs.datastax.com/en/kafka/doc/kafka/kafkaHowMessages.html> (dostęp 2023-01-22)
- [20] Dokumentacja Docker-Compose. *Overview* – <https://docs.docker.com/compose> (dostęp 2023-01-19)

- [21] Dokumentacja Google Cloud. *Event-driven architecture with Pub/Sub* – <https://cloud.google.com/solutions/event-driven-architecture-pubsub> (dostęp 2022-12-06)
- [22] Dokumentacja Kafki. *APIS* – <https://kafka.apache.org/documentation/#api> (dostęp 2022-12-10)
- [23] Dokumentacja Kafki. *Broker Configs* – <https://kafka.apache.org/documentation/#brokerconfigs> (dostęp 2023-01-16)
- [24] Dokumentacja Kafki. *Documentation* – <https://kafka.apache.org/documentation> (dostęp 2023-01-22)
- [25] Dokumentacja NSQ. *Design* – <https://nsq.io/overview/design.html> (dostęp 2023-01-09)
- [26] Dokumentacja Prometheusa. *Exposition Formats* – https://prometheus.io/docs/instrumenting/exposition_formats (dostęp 2023-01-14)
- [27] Dokumentacja RabbitMQ. *RabbitMQ Tutorials* – <https://www.rabbitmq.com/getstarted.html> (dostęp 2023-01-14)
- [28] Dokumentacja RabbitMQ. *Which protocols does RabbitMQ support?* – <https://www.rabbitmq.com/protocols.html> (dostęp 2023-01-03)
- [29] Dokumentacja RocketMQ. *Domain Model* – <https://rocketmq.apache.org/docs/domainModel/01main> (dostęp 2023-01-22)
- [30] Dokumentacja RocketMQ. *Push, Pull and Long Polling* – <https://rocketmq.apache.org/docs/4.x/consumer/11concept2#push-pull-and-long-polling> (dostęp 2023-01-15)
- [31] Dokumentacja RocketMQ. *Quick Start* – <https://rocketmq.apache.org/docs/quickStart/02quickstart> (dostęp 2023-01-15)
- [32] Dokumentacja RocketMQ. *Simple Message Sending* – <https://rocketmq.apache.org/docs/4.x/producer/05message1> (dostęp 2023-01-15)
- [33] Dokumentacja RocketMQ. *Why choose RocketMQ* – <https://rocketmq.apache.org/docs> (dostęp 2023-01-17)
- [34] Fowler, Martin. 2014. *Microservices* – <https://martinfowler.com/articles/microservices.html> (dostęp 2023-01-01)
- [35] Huerta, David de Torres. 2021. *How to monitor Amazon SQS with Prometheus* – <https://sysdig.com/blog/monitor-amazon-sqs-prometheus> (dostęp 2023-01-15)
- [36] Jia, Feng, Wang Xiaorui. 2017. *From Alibaba to Apache: RocketMQ's Past, Present, and Future* – <https://www.infoq.com/articles/alibaba-apache-rocketmq> (dostęp 2023-01-03)
- [37] Johansson, Lovisa. 2019. *Part 1: RabbitMQ for beginners - What is RabbitMQ?* <https://www.cloudamqp.com/blog/part1-rabbitmq-for-beginners-what-is-rabbitmq.html> (dostęp 2023-01-22)
- [38] Khandelwal, Shishir. 2021. *What is Prometheus Grafana stack?* – <https://www.whizlabs.com/blog/prometheus-grafana-stack> (dostęp 2023-01-09)
- [39] Korab, Jakub. 2017. *Understanding Message Brokers* - ISBN 9781491981535
- [40] Livens, Jay. 2022. *What is a Message Queue? How an Observability Platform Eases Message Queue Monitoring* – <https://www.dynatrace.com/news/blog/what-is-a-message-queue> (dostęp 2023-01-01)
- [41] Oficjalna strona RabbitMQ – <https://www.rabbitmq.com> (dostęp 2023-01-22)
- [42] Osin, German. 2021. *Overview of UI Tools for Monitoring and Management of Apache Kafka Clusters* – <https://towardsdatascience.com/overview-of-ui-tools-for-monitoring-and-management-of-apache-kafka-clusters-8c383f897e80> (dostęp 2023-01-16)
- [43] Rasmussen, Peter Daugaard. 2018. *Should you use MQ/Brokers or Web services? And how are they different?* – <https://peterdaugaardrasmussen.com/2018/08/27/should-you-use-mq-brokers-or-web-services-and-how-they-are-different> (dostęp 2023-01-01)

- [44] Riggins, Jennifer. 2022. *Prometheus at 10: What's Been Its Impact on Observability?* – <https://thenewstack.io/prometheus-at-10-whats-been-its-impact-on-observability> (dostęp 2023-01-02)
- [45] Walker, James. 2021. *What Is Prometheus and Why Is It So Popular?* – <https://www.howtogeek.com/devops/what-is-prometheus-and-why-is-it-so-popular> (dostęp 2023-01-02)
- [46] Zestawienie systemów kolejkowych G2. *Best Message Queue (MQ) Software* – <https://www.g2.com/categories/message-queue-mq> (dostęp 2022-12-05)
- [47] Zestawienie systemów kolejkowych Peerspot. *Best Message Queue (MQ) Software* – <https://www.peerspot.com/categories/message-queue-mq-software> (dostęp 2022-12-05)
- [48] Zestawienie systemów kolejkowych Stackshare. *What are the best Message Queue Tools?* – <https://stackshare.io/message-queue> (dostęp 2022-12-05)