POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Dawid Witaszek
Nr albumu 26167

Różne sposoby realizacji komunikacji w systemach korzystających z mikroserwisów

Praca magisterska
Dr. Inż. Mariusz Trzaska

Warszawa, miesiąc, 2023

Streszczenie

Niniejsza praca magisterska dotyczy porównania różnych sposobów realizacji komunikacji w architekturze mikroserwisów. Komunikacja pomiędzy poszczególnymi usługami jest kluczową kwestią przy projektowaniu systemu w oparciu o tę architekturę. Jedną z ważnych decyzji jest wybranie protokołu do komunikacji. Istnieje wiele dostępnych rozwiązań umożliwiających komunikacje, które znacznie różnią się od siebie. W pracy podjęto próbę kompleksowego spojrzenia na technologie umożliwiające wymianę danych. Technologie, które wybrano do porównania to *http*, *graphql*, *kafka* oraz *grpc*. W celu porównania różnych technologii przygotowano aplikację społecznościową w czterech wariantach. Dostarczają one gotowe rozwiązanie, które może konkurować z rozwiązaniami komercyjnymi, oraz stanowi bazę porównawczą wybranych technologii. Poniższa praca stanowi źródło informacji potrzebnych do zaprojektowania systemu opartego o architekturę mikroserwisów oraz proponuje sposób porównania wybranych technologii.

Słowa kluczowe: porównanie technologii, porównanie protokołów sieciowych, architektura aplikacji, mikroserwisy, projektowanie systemów rozproszonych, systemy rozproszone

Spis treści

1. Wstęp

Rozpowszechnienie internatu zwiększyło wymagania aplikacji. Prosta architektura monolityczna przestała spełniać wymagania postawione przez rynek. Aplikacje musiały być łatwo skalowalne, niezawodne i rozwijane w międzynarodowych zespołach. W celu spełnienia tych wymagań powstał szereg rozwiązań określanych jako systemy rozproszone. W wyniku ewolucji tych systemów wyklarowało się kilka wzorców architektonicznych, z czego najbardziej popularne w ostatnich latach stały się mikroserwisy. Według badań przeprowadzonych przez IBM [1] aż 48% aplikacji stworzonych w latach 2019-2021 powstało w oparciu o mikroserwisy. Jednak rozwiązanie to nie jest idealne, 54% z firm [1], które wzięły udział w badaniu stwierdziło, że złożoność mikroserwisów stanowi znaczne wyzwanie. Jednym z wyzwań jest zaprojektowanie komunikacji pomiędzy mikroserwisami. Istnieje wiele technologii służących do komunikacji między usługami. Technologie te różnią się od siebie wieloma cechami, mimo rozwiązywania tego samego problemu. Rozbieżności pomiędzy technologiami są widoczne w sposobie ich użycia, wydajności, wpływie na architekturę oraz doświadczeniu użytkownika.

Początek pracy jest przeznaczony wprowadzeniu w tematykę mikroserwisów oraz wybranych technologii, co pozwoli na zrozumienie wyników przeprowadzonej analizy wybranych technologii. Rozwiązania, które wybrano do porównania to *http*, *graphql*, *kafka* oraz *grpc*. W celu praktycznego porównania technologii przygotowano aplikację społecznościową opartą o *Kotlin*, szkielet aplikacyjny *Spring*, *Keycloak* oraz *MongoDB*. Technologie służące do przygotowania aplikacji zostały wybrane z uwagi na swoją popularność wśród osób korzystających z mikroserwisów. Według statystyk *JetBrains* aż 41% [2] osób implementujących mikroserwisy korzysta z *Javy*. *Spring* jest najpopularniejszym szkieletem aplikacyjnym wykorzystywanym w *Javie*. W celu przyśpieszenia pracy nad aplikacją jako język wybrano *Kotlin*, który jest językiem kompilowanym do kodu *Javy*, tym samym korzystający z jej bibliotek. Korzystając z wybranego języka programowania przyjrzano się sposobowi w jaki implementuje się komunikacje z użyciem danego rozwiązania.

Poniższa praca dyplomowa porusza wymienione wyżej kwestie oraz omawia różnice i podobieństwa występujące pomiędzy wybranymi technologiami. Analiza została podzielona na kilka perspektyw, z których będą oceniane poszczególne technologie. Ocenie będzie podlegać wydajność, wpływ na architekturę, złożoność, wymagania infrastrukturalne, sposób użytkowania technologii oraz jej dostępność w innych językach programowania. W ramach podsumowania analizy została podjęta próba znalezienia kompromisu przy wyborze technologii umożliwiającej komunikacje pomiędzy mikroserwisami.

2. Wstęp do architektury mikroserwisów

Mikroserwisy [3] są podejściem do tworzenia systemów rozproszonych. Polega to na stworzeniu niewielkich usług, które współpracują ze sobą. Mikrousługa powinna być skonstruowana w taki sposób, aby dało się ją modyfikować oraz wdrażać niezależnie od innych usług. Usługa jest modelowana wokół domeny biznesowej. Modelem domeny nazywamy model koncepcyjny, który przedstawia podmioty składające się na dziedzinę, w której działa proces biznesowy. Usługa zbudowana wokół domeny hermetyzuje funkcjonalność i udostępnia je innym usługom. Kluczowym elementem jest zakreślenie granic domeny, czyli odpowiedzialności pojedynczej usługi.

Mikrousługa powinna stanowić czarną skrzynkę dla innych usług składających się na ten sam system. Funkcje usługi są udostępnione dla innych części systemów za pośrednictwem protokołów sieciowych np. *http*. Dzięki takiemu zabiegowi klienci mogą korzystać z usług bez znajomości szczegółów implementacyjnych. Docelowo mikroserwisy nie powinny współdzielić bazy danych, zamiast tego każda usługa korzysta ze swojej bazy.

Mikroserwisy szczególnie dobrze sprawdzają się w organizacjach ukierunkowanych w stronę luźnego powiazania usług w celu stworzenia samowystarczalnych zespołów. Takie zespoły mogą dostarczać funkcjonalności tworzonych ściśle według wymagań klienta, będąc niezależnymi od siebie. Ogromną zaletą mikroserwisów jest także elastyczność w doborze stosu technologicznego z jakiego korzystają poszczególne usługi. Mikroserwisy posiadają jednak jedną dużą wadę jaką jest złożoność, którą wprowadzają. Złożoność systemów rozproszonych jest problematyczna nawet dla najbardziej doświadczonych programistów.

Ważnym pojęciem przy projektowaniu mikrousług jest *Domain Driven Desing* [4]. Jest to sposób projektowania aplikacji z perspektywy domeny biznesowej. Analizując wymagania biznesowe możemy przydzielić poszczególne funkcjonalności do kontekstu, z którym są związane. Dzięki podziałowi aplikacji na logiczne części jesteśmy w stanie wyodrębnić poszczególne mikroserwisy, tak aby zminimalizować wymianę informacji pomiędzy nimi. Dzięki nakreśleniu wyraźnych i stabilnych granic mikrousług można uzyskać system, który charakteryzuje się wysoką spójnością pomimo autonomicznych usług.

Mikroserwisy posiadają kilka wyróżniających:

- Niezależne wdrożenia – stworzenie luźno powiązanych usług pozwala na niezależne wdrożenie każdej z nich. Wprowadzając zmiany do jednego serwisu nie trzeba aktualizować całego systemu, a jedynie jego małą część.
- Rozmiar – sam przedrostek „mikro” sugeruje, że każdy z serwisów powinien być względnie mały. Docelowy rozmiar usługi jest zależny od nas i nie jest mierzony ilością kodu, a złożonością interfejsu. Pojedyncza usługa powinna reprezentować jedną odpowiedzialność biznesową, podobnie do zasady *Single Responsibility Principle* [5] z programowania obiektowego.
- Elastyczność – mikroserwisy oferują swobodę zarówno pod kątem skalowania, organizacji zespołów i doboru technologii. Stos każdej usługi jest dobierany oddzielnie, dzięki czemu możemy dobrać narzędzie adekwatne do problemu. Jeżeli tylko jedna funkcjonalność systemu jest bardziej obciążona, to możemy skalować mikroserwis odpowiedzialny za tę funkcjonalność. Organizacja zespołów jest uproszczona, ponieważ te mogą skupić się na rozwijaniu małej aplikacji, która współgra z innymi.
- Odporność na błędy – podczas awarii części systemu reszta aplikacji jest nadal w stanie oferować ograniczoną funkcjonalność dla klientów końcowych.

- Komponowalność – każda usługa może korzystać z innych usług jak z zewnętrznego *API*, co pozwala na wielokrotne wykorzystanie funkcjonalności.

3. Praktyczny projekt oprogramowania społecznościowego

Na potrzeby niniejszej pracy przygotowano serwerową część aplikacji społecznościowej. Stworzony system posiada szereg funkcjonalności kojarzonych z tego typu aplikacjami. Użytkownicy mają możliwość budowania sieci znajomych, umieszczania postów, komentowania, oceniania oraz zakładania grup tematycznych. Ważnym elementem aplikacji społecznościowej jest system autoryzacji użytkowników, który także został utworzony. System miał spełniać nie tylko wymagania funkcjonalne, ale także szereg wymagań niefunkcjonalnych, stanowiących główną trudność.

3.1. Cel projektu

Aplikacja zaimplementowana na potrzeby pracy miała posłużyć jako przykład wykorzystania architektury mikroserwisów oraz umożliwić porównanie wariantów komunikacji. Stworzono kilka wariantów tej samej aplikacji, wzbogaconych o mierniki wydajności, aby rzetelnie porównać technologie. Celem stworzenia aplikacji nie była jej złożoność, a umożliwienie przeprowadzenia analizy. Dołożono starań, aby współdzielić kod aplikacji w taki sposób, aby różnice w wydajności nie wynikały z innej implementacji. Dlatego podczas budowy systemu zastosowano szereg praktyk i wzorców pozwalających na korzystanie z tej samej bazy kodu we wszystkich wariantach. Nie udało się jednak uniknąć przepisania poszczególnych fragmentów programu, co wynika z różnić pomiędzy technologiami.

3.2. Opis projektu

Projekt posiada kilka wariantów, z czego każdy z nich korzysta z innej metody komunikacji pomiędzy poszczególnymi usługami. Wszystkie warianty realizują taką samą funkcjonalność. Z uwagi na cel pracy, którym jest analiza kilku rozwiązań przygotowano część serwerową aplikacji. Wpływ każdej z technologii na klienta korzystającego z aplikacji został zbadany za pomocą narzędzia *Postman* lub zwięzłych implementacji klienta. Aby zachować realia w projekcie zostały użyte najpopularniejsze technologie. Aplikacja napisana jest z użyciem języka *Kotlin*, który wykorzystuje biblioteki *Javy* i jest do niej kompilowany. Jako bazę danych wykorzystano *MongoDB*. Część serwerowa, jak i integracja z protokołami komunikacyjnymi została stworzona przy pomocy szkieletu aplikacyjnego *Spring*. Dodatkowo w celu ułatwienia pracy nad projektem użyto narzędzia do wirtualizacji *Docker*. Każdy wariant posiada zbliżony stos technologiczny, a jedyne różnice, które pojawiają się pomiędzy nimi wynikają z wykorzystanego protokołu komunikacyjnego.

Wszystkie technologie, które zostały poddane analizie musiały posiadać ugruntowane miejsce na rynku. Jako technologie pomocnicze wybrano te, które posiadają szereg funkcjonalności pozwalających na implementację mikroserwisów przy ich użyciu. Szkielet aplikacyjny musiał przede wszystkim posiadać moduły pozwalające na integrację z kilkoma protokołami komunikacyjnymi oraz posiadać opinie zaufanego narzędzia. Próbowano uniknąć sytuacji, w której zła ocena technologii będzie wynikać z biblioteki do integracji. Przy użyciu sprawdzonych i popularnych technologii umieszczone fragmenty kodu stanowią nie tylko przykład, ale i solucję do tworzenia oprogramowania w oparciu o dany protokół komunikacyjny.

Tematyka projektu została wybrana tak, aby zbudować system bliski realnym rozwiązaniom. Aplikacja społecznościowa posiada dużą ilość funkcjonalności, które da się rozdzielić na poszczególne mikrousługi takie jak grupy, profile, posty czy komentarze.

3.3. Wymagania projektowe

Projekt spełnia szereg wymagań funkcjonalnych i niefunkcjonalnych postawionych na początku pracy:

Tabela 1. Spis wymagań dotyczących aplikacji

Nr	Opis wymagania	Rodzaj	Czy zrealizowano
1	System powinien być oparty o stos *Javy*	Niefunkcjonalne	Tak
2	API ma udostępniać zasoby w postaci *JSON*	Niefunkcjonalne	Tak
3	Aplikacja ma spełniać zasady *REST*, o ile to możliwe	Niefunkcjonalne	Nie udało się zrealizować dla *Kafki*
4	Bazą danych ma być *MongoDB*	Niefunkcjonalne	Tak
5	Całość implementacji ma być oparta o szkielet aplikacyjny *Spring*	Niefunkcjonalne	Tak
6	Dostęp do aplikacji powinien być zastrzeżony tylko dla osób zalogowanych	Funkcjonalne	Tak
7	Aplikacja powinna korzystać z protokołu *OAuth2* do autoryzacji	Niefunkcjonalne	Tak
8	Autentykacja użytkownika ma być obsłużona przez *OpenID*	Niefunkcjonalne	Tak
9	Autoryzacja i autentykacja powinna zostać wydzielona do osobnej usługi *IAM* np. *Keycloak*	Niefunkcjonalne	Tak
10	Dostęp do poszczególnych usług ma przebiegać przez *API Gateway*	Niefunkcjonalne	Tak
11	Poszczególne moduły mają być budowane za pomocą *Gradle*	Niefunkcjonalne	Tak
12	Każdy z mikroserwisów powinien być skonteneryzowany	Niefunkcjonalne	Tak

13	System powinien składać się z minimum 3 mikrousług	Niefunkcjonalne	Tak
14	Aplikacja powinna implementować *SSO*, tak aby każdy z mikroserwisów nie musiał logować się z osobna	Niefunkcjonalne	Tak
15	Aplikacja powinna posiadać bardziej złożone operacje, które będą wymagać komunikacji pomiędzy mikroserwisami	Niefunkcjonalne	Tak
16	Każda z technologii użytych do komunikacji ma być użyta w optymalny sposób	Niefunkcjonalne	Tak
17	Implementacja funkcjonalności ma być współdzielona pomiędzy wariantami	Niefunkcjonalne	Tak
18	Aplikacja ma korzystać z architektury mikrousług	Niefunkcjonalne	Tak
19	Poszczególne mikroserwisy mają być zbudowane na bazie architektury heksagonalnej, aby odseparować implementacje od technologii	Niefunkcjonalne	Tak
20	Każdy mikroserwis powinien posiadać własną bazę danych. Dane nie mogą być współdzielone pomiędzy mikroserwisami	Niefunkcjonalne	Tak
21	Każdy z wariantów aplikacji ma oferować te same funkcjonalności z minimalnie zmienionym interfejsem	Niefunkcjonalne	Tak
22	Należy uniknąć podawania ręcznie adresów poszczególnych mikroserwisów, aby umożliwić komunikację	Niefunkcjonalne	Tak
23	Kontekst dla każdej usługi ma być wydzielony za pomocą strategicznego *Domain Driven Design*	Niefunkcjonalne	Tak
24	Każdy mikroserwis, o ile to możliwe, powinien móc przełączać się pomiędzy protokołami za pomocą profilu	Funkcjonalne	Tak
25	Infrastruktura ma być wystawiana za pomocą narzędzia *Docker Compose*	Funkcjonalne	Tak
26	System powinien być dostępny w czterech wariantach, osobno dla *kafki*, graphql, *grpc*, *http*	Funkcjonalne	Tak
27	Każdy z użytkowników powinien posiadać zasoby powiązane tylko z jego kontem/profilem	Funkcjonalne	Tak

28	Użytkownik może modyfikować jedynie zasoby, których jest właścicielem	Funkcjonalne	Tak
29	Użytkownik może dodawać posty, które są widoczne dla jego znajomych	Funkcjonalne	Tak
30	Użytkownicy mogą komentować posty	Funkcjonalne	Tak
31	Użytkownicy mogą odpowiadać na swoje komentarze, tworząc ciągi konwersacji	Funkcjonalne	Tak
32	Użytkownicy mogą tworzyć grupy	Funkcjonalne	Tak
33	Treści dotyczące grupy mogą być widoczne tylko dla jej członków	Funkcjonalne	Tak
34	Użytkownicy mają możliwość obserwowania innych profili	Funkcjonalne	Tak
35	Użytkownik może wygenerować dla siebie *feed*, czyli zbiór postów, które pojawiły się na profilach które śledzi i grupach, do których należy	Funkcjonalne	Tak
36	Właściciele grupy mogą ją edytować	Funkcjonalne	Tak
37	Właściciel grupy może mianować administratorem innego użytkownika	Funkcjonalne	Tak

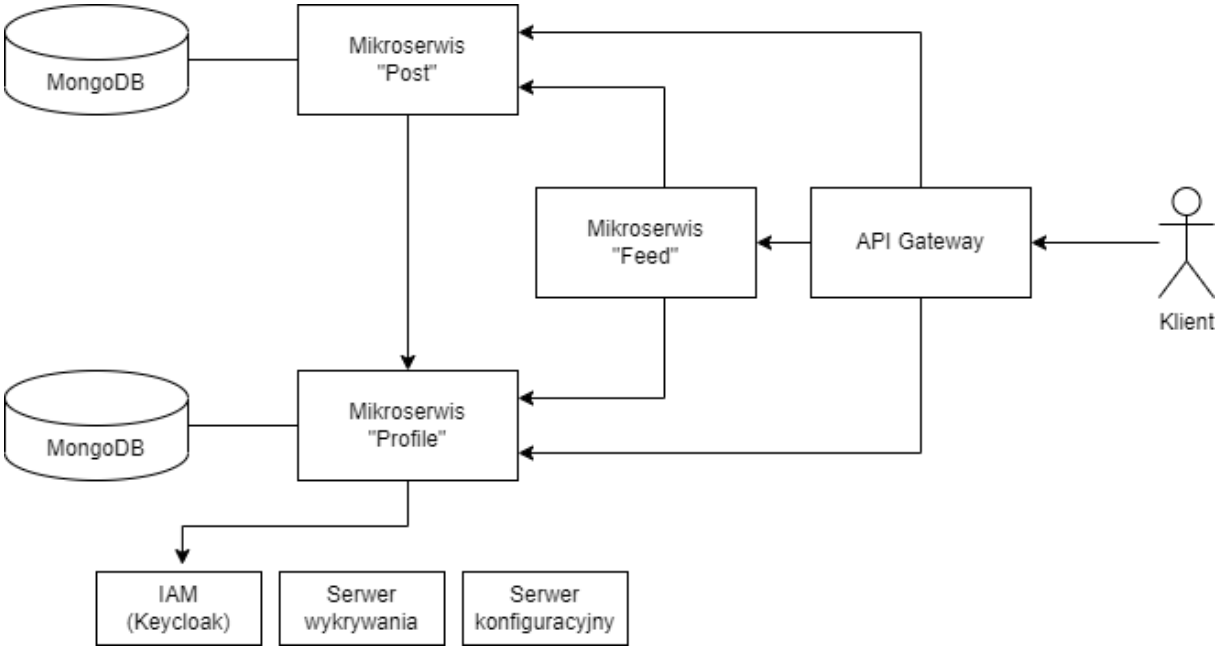
3.4. Architektura projektu

Projekt jest zbudowany zgodnie z architekturą mikroserwisów. Podczas implementacji wzięto pod uwagę wszystkie zasady dotyczące budowania mikrousług, przez co podjęto szereg decyzji przy budowaniu aplikacji. Autoryzacja i autentykacja została wydzielona do *Keycloak*. Każda z usług posiada zbiór funkcjonalności pogrupowanych przy użyciu podejścia *Domain Driven Design*. Architektura w niewielkim stopniu różni się pomiędzy poszczególnymi wariantami, co wynika z charakterystyki protokołu komunikacyjnego. Różnice polegają jednak na infrastrukturze, która jest potrzebna do prawidłowego działania aplikacji.

Poszczególne usługi są zbudowane na bazie architektury heksagonalnej [6]. Każda z mikrousług posiada kilka modułów, które w zależności od ustalonego profilu są załączane do aplikacji lub nie. Podczas zmiany profilu wymieniona jest zewnętrzna warstwa tak zwanych adapterów. Dzięki zastosowaniu architektury heksagonalnej w prosty sposób możemy współdzielić kod pomiędzy wariantami aplikacji bez potrzeby przepisywania całości usługi. Różnica pomiędzy poszczególnymi implementacjami wynika z różnych warstw adapterów. Warstwa adapterów jest to najbardziej zewnętrzna warstwa aplikacji, która bezpośrednio zintegrowana jest różnymi technologiami. Pozostałe

warstwy, czyli warstwa aplikacji oraz warstwa domeny pozostają bez zmiany. Warstwa domeny jest to centralna część aplikacji, która posiada minimum zależności od technologii, w tym przypadku jest to jedynie język programowania oraz narzędzie do budowania modułu. Warstwa aplikacyjna natomiast implementuje poszczególne przypadki użycia oraz udostępnia projekcje obiektów domeny.

3.5. Architektura w przypadku protokołu *http*



Rysunek 1. Architektura aplikacji w wariancie *http*. Komunikacja pomiędzy serwerami wykonania, konfiguracyjnym oraz *IAM* zostały pominięte dla klarowności schematu

Najłatwiej przybliżyć architekturę aplikacji posługując się wariantem *http*, który posiada minimalną ilość dodatkowych elementów. Po zapoznaniu się ze schematem aplikacji można dojść do dwóch błędnych wniosków: aplikacja posiada tzw. wąskie gardło w postaci *API Gateway* oraz posiada sprzężenia. Przed *API Gateway* w warunkach produkcyjnych powinien znajdować się *LoadBalancer*, który został na schemacie pominięty, ponieważ nie wnosi nic z perspektywy celu pracy. Mimo wielu połączeń pomiędzy usługami na schemacie komunikacja pomiędzy mikroserwisami w rzeczywistości jest ograniczona do kilku przypadków. Istnieje możliwość usunięcia połączenia pomiędzy serwisem *Feed* a *Profile,* jednak nie została ona zaimplementowana celowo na potrzebę pracy. Aplikacja składa się z kolejnych elementów:

- Mikroserwis „*Post*" – usługa jest odpowiedzialna za usługę żądań i wykonania operacji związanych z edycją, tworzeniem i udostępnianiem postów, grup oraz komentarzy.
- Mikroserwis „*Feed*" – jest to usługa, która przygotowuje zbiór postów dla użytkownika bazując na sieci obserwowanych profili oraz grup, do których należy
- Mikroserwis „*Profile*" – odpowiada za tworzenie i edycję profili oraz za zarządzanie siecią społecznościową użytkownika. Stanowi także pośrednika pomiędzy klientem a systemem *IAM*. Użytkownik chcąc zmienić email lub hasło musi odwołać się do *IAM* za pośrednictwem tej usługi

- *API Gateway* – jest implementacją wzorca stosowaną w systemach rozproszonych [7]. Stanowi on pojedynczy punkt wejścia dla żądań wysyłanych do aplikacji przez klienta. Dzięki zastosowaniu tego wzorca klient ma złudzenie łączenia się z monolitem i jest niezależny od zmian w architekturze. Dodatkowo *API Gateway* oddziela komunikację zewnętrzną (między serwerem a klientami) od komunikacji wewnętrznej (pomiędzy usługami)
- *IAM – Identity and Access Management*- jest to usługa, która pozwala na kontrolowanie dostępu do zasobów systemu. Dzięki osobnemu serwerowi *IAM* można scentralizować zarządzanie uprawnieniami oraz kontrolować autoryzacje i autentykacje w całej aplikacji. Jako implementacje *IAM* wykorzystano narzędzie *Keycloak*
- Serwer wykrywania – z angielskiego *Discovery Server* służy do rejestrowania instancji usług, które są obecne w infrastrukturze. Usługa chcąca skomunikować się z inną dostaje adres do instancji usługi docelowej od serwera wykrywania. Dodatkowo serwer wykrywania pełni funkcje *Load Balancer'a* rozdzielając ruch pomiędzy kilkoma zarejestrowanymi instancjami jednej usługi
- Serwer konfiguracyjny – startująca usługa odwołuję się do tej usługi w celu pobrania atrybutów konfiguracyjnych
- *MongoDB* – *MongoDB* jest nierelacyjną bazą danych w której są zapisywane dane mikrousług Profile i Post
- Klient – został umieszczony na schemacie, aby pokazać jak potencjalny użytkownik łączy się z aplikacją

Strzałki zamieszczone na schemacie demonstrują przepływ komunikacji w systemie. Grot strzałki pokazuje na usługę, która otrzymuje żądanie od usługi, która wysyła żądanie. W aplikacji jest kilka przypadków, w których poszczególne serwisy łączą się ze sobą:

- *Post/Profile* – kiedy powstaje grupa lub zostaje usunięta usługa *„Post"* odwołuję się do usługi *„Profile"* w celu przekazania informacji o potencjalnej zmianie w sieci społecznościowej użytkowników
- *Feed/Profile* – aplikacja w celu przygotowania zbioru postów dla użytkownika odwołuje się do usługi *„Feed"* aby poznać sieć społecznościową użytkownika.
- *Feed/Post* – aplikacja znając już sieć społecznościową użytkownika buduje zapytanie do serwisu obsługującego posty i komentarze w celu pobrania informacji
- *Profile/IAM* – kiedy nowy użytkownik się rejestruje robi to za pośrednictwem usługi *„Profile"*. Usługa najpierw waliduje dane dotyczące profilu społecznościowego, a następnie tworzy użytkownika w serwerze *IAM*. Jeżeli serwer *IAM* poprawnie utworzy użytkownika zostaje stworzony nowy profil.

Oprócz wymienionych wyżej połączeń istnieją także inne pomiędzy *API Gateway*, a poszczególnymi usługami, jednak mają one charakter przekazania żądania. Z perspektywy tematu pracy najważniejszymi połączeniami są te pomiędzy *„Feed*" a *„Profile"* oraz *„Post"* i pomiędzy *„Post"* a *„Profile"*. Oba te połączenia pokazują inną charakterystykę. W przypadku komunikacji pomiędzy *„Feed"* a *„Profile"* następuje prośba o wykonanie operacji, co może doprowadzić do sprzężenia w niektórych technologiach branych pod uwagę. Natomiast wymiana informacji pomiędzy *„Post"* a *„Profile"* ma charakter zdarzenia, gdzie jedna usługa nie musi być pewna, że druga odebrała wiadomość. Cechy te mają bardzo duże znaczenie w dalszej części pracy.

3.6. Technologie oraz narzędzia wykorzystane przy implementacji

Podczas implementacji przykładowego systemu wykorzystano szereg technologii, które są wykorzystywane komercyjnie podczas pracy w oparciu o architekturę mikroserwisów. Pominięto

technologie służące do komunikacji, ponieważ opisano je szczegółowo w dalszej części pracy zajmującej się analizą porównawczą.

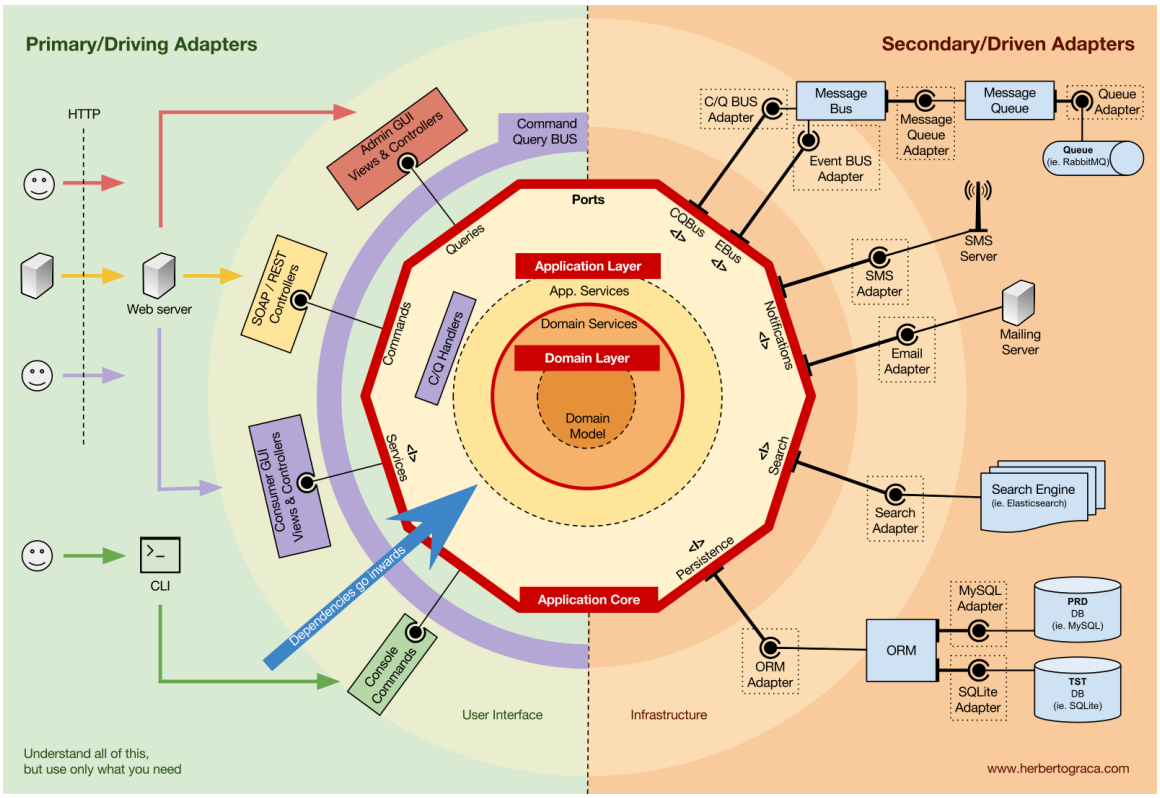
- ***Kotlin*** [8] – jest to wysokopoziomowy, między-platformowy, statycznie typowany język programowania ogólnego przeznaczenia. *Kotlin* został zaprojektowany z myślą o całkowitej kompatybilności z *Javą*. Został wyposażony w dostosowaną wersję *JVM*, która wykorzystuje standardowe biblioteki *Javy*. Dzięki takiemu podejściu możliwe jest stworzenie aplikacji korzystającej z bibliotek *Javy* w oparciu o zwięzły język programowania.
- ***Gradle*** [9] – jest to narzędzie służące automatyzacji budowania projektu programistycznego. Umożliwia także tworzenie i zarządzanie modułami aplikacji. Dzięki możliwości pisania użytkowych skryptów w oparciu o język kotlin za pomocą tego narzędzia można zautomatyzować wiele aspektów pracy nad projektem takich jak testowanie, kompilacja bibliotek, kompilacja schematów *protobuf* i wiele więcej.
- ***Spring Boot*** [10] – jest to nakładka na szkielet aplikacyjny *Spring* umożliwiająca łatwą konfigurację projektu. Platforma ta oferuje szereg rozwiązań, które są wykorzystane w projekcie. Dostarcza ona implementacji serwera aplikacyjnego, biblioteki do przetwarzania zapytań, kontener *IoC* (*Inversion of Control*), biblioteki do łączenia się z bazą *MongoDB*, narzędzia do obsługi logów aplikacji i wiele mniejszych udogodnień. Jej główną zaletą jest posiadanie sprawdzonych integracji do wszystkich technologie, które analizowano
- ***MongoDB*** [11] – jest to dokumentowa baza danych *NoSQL*. Działa w oparciu o obiekty przypominające format *JSON*, jednak będące przechowywane w formie binarnej *BSON*. Jedną z wad relacyjnych baz danych było zmapowanie tabel baz danych na szereg klas w języku programowania. Problem ten rozwiązywano poprzez znajdywanie kompromisów oraz implementowania doraźnych optymalizacji. *MongoDB* dzięki zastosowaniu modelu danych przypominającym mapy rozwiązano problem z translacją klas na tabele relacyjnych bazy danych. Dokumentowa baza posiada kolekcje, które zawierają obiekty opisane wartościami klucz-wartość, a które mogą posiada listy i zagnieżdżenia.
- ***Docker*** [12] – technologia ta umożliwia konteneryzację. Termin konteneryzacji oznacza obudowanie procesu aplikacji odchudzona wirtualną maszyną udostępniającą system operacyjny specjalnie na potrzeby tego procesu. Oprócz konteneryzacji kontener *Docker'a* stanowi format pakowania w ustandaryzowanym formacie oraz pozawala na szybkie i nienaganne działanie na różnych środowiskach. *Docker* stanowi także platformę programistyczną, która umożliwia programistom komfortowe tworzenie, wdrażanie oraz testowanie aplikacji.
- ***Docker Compose*** [13] – jest to narzędzie będące rozszerzeniem *Docker'a,* które używa formatu *YAML* do skonfigurowania kontenerów aplikacji oraz uruchamia wszystkie procesy w nim zawarte. Narzędzie pozwala także na zarządzanie grupą działających kontenerów z poziomu powłoki systemu. Używanie *Docker Compose* znacznie poprawia produktywność przy pracy nad systemami rozproszonymi oraz umożliwia zapisanie ich konfiguracji do pliku.
- ***Keycloak*** [14] – jest to otwarto-źródłowe oprogramowanie dostarczające implementacji *Single Sign On* oraz *Identity and access management*.
- ***Postman*** [15] – aplikacja ułatwiająca projektowanie, budowanie oraz testowanie *Application Programming Interface*
- ***Python*** [16] – jest wysokopoziomowym językiem programowania, którego główne cechy to zwięzłość i łatwość w zrozumieniu kodu. Język posiada dużą bazę bibliotek, które są przydatne w pisaniu skryptów. Narzędzie to było wykorzystane w czasie tworzenia aplikacji do testowania niektórych narzędzi oraz automatyzowania powtarzalnych obowiązków programisty.
- ***Jmeter*** [17] – narzędzie pozwalające na przeprowadzenie testów wydajnościowych aplikacji. Jest ono bardzo proste w działaniu i polega na zadeklarowaniu sekwencji operacji, które trzeba wykonać w oddzielnych wątkach. Ilość wątków i częstotliwość wywołani jest określona przez użytkownika.

3.7. Zastosowane wzorce umożliwiające współdzielenie kodu

W śród wymagań niefunkcjonalnych znalazło się współdzielenie kodu tak, aby zapewnić wiarygodność wyników. Zbyt kosztownym byłoby także implementowanie tej samej logiki aplikacji czterokrotnie. Kluczowym było wykorzystanie praktyk i wzorców które pozwolą na zbudowanie łatwo rozszerzalnego oprogramowania, które łatwo modyfikować. Kolejnym wymogiem było zapewnienie modułowości oraz konfiguracji, aby w prosty sposób przełączyć się pomiędzy poszczególnymi wariantami systemu.

3.7.1 Architektura heksagonalna

Głównym wzorcem umożliwiającym oddzielenie kodu domeny (opisującego logikę aplikacji) jest architektura heksagonalna (zwana także architekturą portów i adapterów lub czystą) [18]. Podejście daje szereg zasad, które trzeba spełnić, aby wyodrębnić „serce" aplikacji, które da się umieścić w różnej obudowie, którą stanowią biblioteki zewnętrzne. Architektura ta nie odnosi się do całości systemu rozproszonego, ale określa, jak są zbudowana poszczególne usługi.



Rysunek 2. Poglądowy schemat architektury heksagonalnej [18]

Najważniejszym elementem architektury jest warstwa domenowa, która jest logiką naszej aplikacji. Celem jest odseparowanie tej logiki od technologii i dostarczenie interfejsu (portów), który umożliwi integracje systemu z różnymi zbiorami technologii. Porty są elementami kodu, które łączą narzędzia z rdzeniem aplikacji jaką jest domena. Natomiast adaptery są jednostkami komunikującymi się z portami, które pozwalają adapterom na użycie logiki zawartej w domenie lub wstrzyknięcia kodu, który będzie używany przez domenę. Port nie jest niczym więcej niż specyfikacją dla zewnętrznych narzędzi określającą kontrakt z rdzeniem aplikacji. W większości przypadków port jest równoznaczny

z interfejsem zawartym w rdzeniu aplikacji, ale może być też grupą interfejsów i *Data Transfer Object*.

3.7.2 Przykład implementacji

Najłatwiej zobrazować opisany wyżej mechanizm na przykładzie bazującym na aplikacji napisanej na potrzeby pracy.

Listing 1. Przykładowy port, stanowiący specyfikacje dostępu do bazy danych

```
interface EntityRepository <E: DDDEntity> {
fun save(e: E): E
fun deleteById(id: UUID)
fun removeAll(ids: List<UUID>)

fun findById(id: UUID): E?
fun findAllByIds(ids: List<UUID>): List<E>
fun existById(uuid: UUID): Boolean
}
```

```
interface CommentRepository: EntityRepository<Comment> {
fun removeAllByPostId(postId: UUID)
}
```

Implementacja portu jest bardzo prosta. Budując rdzeń aplikacji pomijamy miejsca w których normalnie użylibyśmy zewnętrznych technologii, a zamiast tego używamy interfejsu. Otrzymany interfejs stanowi specyfikację dla naszego portu.

Listing 2. Adapter wykorzystujący port aplikacji

```
@Component
class CommentRepositoryImpl(private val commentRepository: CommentJpaRepository):
CommentRepository {

override fun save(e: Comment): Comment {
return commentRepository.save(e.toCommentTable()).toComment()
    }

override fun deleteById(id: UUID) {
commentRepository.deleteById(id)
    }

override fun removeAll(ids: List<UUID>) {
commentRepository.deleteAllById(ids)
    }

override fun findById(id: UUID): Comment? {
return commentRepository.findById(id)
.map { it.toComment() }
.orElse(null)
    }
```

```
override fun findAllByIds(ids: List<UUID>): List<Comment> {
return commentRepository.findAllById(ids)
.map { it.toComment() }
    }

override fun existById(uuid: UUID): Boolean {
return commentRepository.existsById(uuid)
    }

override fun removeAllByPostId(postId: UUID) {
commentRepository.removeAllByPostId(postId)
    }
}
```

Otrzymaną implementację musimy następnie rozdzielić pomiędzy moduły. Potrzebujemy 2 moduły, które będą posiadały odmienną charakterystykę. Pierwszy moduł będzie przeznaczony dla naszej domeny i będzie zawierał interfejs *EntityRepository* i *CommentRepository*. Moduł domeny nie może zawierać zależności do bibliotek, taki moduł może być związany jedynie z językiem programowania i z narzędziem służącym do budowania modułów. Drugim modułem jest nasza aplikacja, która posiada zależność do zewnętrznych bibliotek oraz do modułu domeny. W ten sposób otrzymaliśmy moduł, który zawiera jądro aplikacji oraz wymienialną powłokę.

3.7.3 Profile aplikacji

Kolejnym elementem umożliwiającym maksymalne współdzielenie kodu pomiędzy czterema różnymi wariantami aplikacji są profile konfiguracyjne. Aplikacja posiada 4 różne profile: *kafa, grpc, graphql* oraz *feign(http),* które to aktywują różne protokoły komunikacji. *Spring* jako platforma dostarcza implementacji profil [19] i która jest scalona z kontenerem *bean'ów*. *Bean'em oznacza* instancje klasy, która jest wykorzystywana w aplikacji na przykład jako kontroler lub obiekt nasłuchujący brokera wiadomości.

Listing 3. Przykład adnotacji określającej przynależność *Bean'a* do profilu

```
@Component
@Profile(„feign”)
class ProfileServiceClientFeign
```

Łącząc podział aplikacji na moduły, profile oraz architekturę heksagonalną otrzymano rozwiązanie pozwalające na uniknięcie potrzeby przepisywania aplikacji. Dodatkowo profile konfiguracyjne zwiększają produktywność pracy nad projektem, ponieważ przełączenie protokołów komunikacji w usługach sprowadza się do ponownego uruchomienia ich ustawiając inny profil konfiguracyjny.

4. Charakterystyka technologii

Technologie brane pod uwagę w analizie zostały wybrane z powodu swojej popularności oraz ich rozpowszechnieniu w warunkach komercyjnych. Dodatkowo każda z technologii posiada inną charakterystykę.

4.1. Protokół *http*

Protokół *http*, czyli *Hypertext Transfer Protocol* [20] jest stylem komunikacji synchronicznej blokującej. Oznacza to, że usługa wysyłająca zapytanie do drugiej będzie czekała na odpowiedź, aż ta nie nastąpi. Z uwagi na to, że mamy gwarancje otrzymania odpowiedzi lub błędu, używając protokołu *http* wybieramy komunikacje typu żądanie – odpowiedź.

Protokół jest rozpowszechniony z uwagi na to, że jest wykorzystywany przez przeglądarkę. Rozwój stron internetowych spowodował duże zainteresowanie tym narzędziem, które zaczęło być wykorzystywane w obszarach niezwiązanych z aplikacjami internetowymi. Według statystyk, aż 82% [21] stron internetowych wykorzystuje ten protokół. Natomiast, aż 81% [22] procent systemów wykorzystujących mikrousługi opiera komunikacje o *http*.

Od strony technicznej protokół *http* został stworzony na potrzeby komunikacji w modelu klient-serwer. Przykładowo przeglądarka internetowa jest klientem, a proces zwracający dane znajduje się na serwerze. Klient wysyła żądanie w celu uzyskania zasobu, natomiast serwer dostarcza zasób lub uruchamia operacje na polecenie klienta. W wyniku wykonania żądania zwracana jest odpowiedź zawierająca zasób (o ile taki istnieje) oraz kod wykonania operacji.

Http posiada także ustandaryzowaną składnię zapytania:

- Linia żądania (z angielskiego *Request Line*) - składa się z kilku elementów metody zapytania, adresu *URL* (*Uniform Resource Locator*) oraz wersji protokołu.

Listing 4. Przykład linii żądania

```
GET /files/home.html HTTP/2
```

- Nagłówki wiadomości (*Headers*) – pary typu klucz-wartość są opcjonalne i zawierają dodatkową wiadomość dla serwera, przeglądarki lub serwerów pośredniczących. Przykładowo popularny nagłówek *Cache-Control* służy do dodania informacji dotyczących polityki trzymania podręcznych danych (*cache*).

Listing 4 Nagłówki ustawiające adres docelowy i wyłączające użycie pamięci podręcznej

```
Host: example.com
Pragma: no-cache
Cache-Control: no-cache
```

- Ciało wiadomości (*Body*) – opcjonalne dane wpisane przez użytkownika. Są dowolnego formatu, natomiast najczęściej jest to format JSON (*JavaScript Object Notation*)
- Metody żądań – linia żądania składa się między innymi z metody, która określa cel żądania.

  Istnieje kilka ustandaryzowanych metod:

- *GET* – żądanie służy do pobierania danych z serwera aplikacyjnego. Zaleca się, aby żądanie nie miało wpływu na dane znajdujące się na serwerze, a jedynie je zwracało. Przykładowo przeglądarka korzysta z zapytania *GET,* gdy pobiera elementy strony
- *POST* - celem żądania jest wywołanie operacji na serwerze lub zapisanie na nim danych zawartych w ciele żądania. Przykładowo *POST* jest używany do dodania komentarza w forum lub subskrypcji lisy mailingowych
- *PUT* - żądanie aktualizuje dane, które znajdują się na serwerze. Różni się od *POST* tym, że lokalizacja zasobu jest wskazana, ponieważ powinna już istnieć
- *PATCH* – żądanie podobnie jak *PUT* służy do aktualizacji zasobu, jednak zasób może być zaktualizowany w części. Przykładowo edytując tylko część pliku zmniejszamy wykorzystanie łącza
- *DELETE* – służy do usuwania zasobu znajdującego się na serwerze
- *HEAD, OPTION, CONNECT, TRACE* – metody służą do implementacji technicznych aspektów aplikacji związanych z bezpieczeństwem

- Status odpowiedzi – odpowiedź serwera *http* różni się od żądania między innymi tym, że posiada status odpowiedzi, który definiuje efekt żądania. Statusów żądania jest kilkadziesiąt, jednak używa się tylko kilkunastu z nich. Statusy dzielimy na grupy:
  - 1XX – informacyjny, oznacza, że żądanie zostało przyjęte, a proces jest kontynuowany
  - 2XX – pozytywny, oznacza, że żądanie zostało przetworzone pozytywnie
  - 3XX – żądanie będzie przekierowane
  - 4XX – żądanie nie zostało przeprocesowane poprawnie, ponieważ klient popełnił błąd
  - 5XX – żądanie nie zostało przeprocesowane poprawnie z powodu błędu na serwerze

Protokół *http* miał kilka wersji, które znacznie różną się od siebie pod względem działania. Obecnie 3 wersje protokołu są ustandaryzowane:

- *HTTP/1.1* - wydany w 1997 roku był wersją, która była używana najdłużej, bo aż 14 lat. Protokół charakteryzował się tym, że działał w oparciu o *TCP*. Niestety zwiększające się prędkości łącza internetowego sprawiły, że protokół zaczął pokazywać swoje niedoskonałości. Problemem stało się otwieranie nowego połączenia z serwerem za każdym razem, kiedy chcieliśmy pobrać jakiś zasób. Dodatkowo niemożliwe było pobieranie kilku zasobów na raz bez otwierania wielu połączeń lub zatrzymanie połączenia w celu pobrania kilku zasobów sekwencyjnie. Problemy te wraz ze wzrostem wagi przeciętnej strony internetowej powodowały wolny czas ładowania.
- *HTTP/2* – problemy pierwszej wersji protokołu doprowadziły do powstania kolejnej usprawnionej wersji. Protokół został ustandaryzowany w 2015 i jest najczęściej używanym standardem w czasie powstawania pracy. *HTTP/2* jest używana przez 40% aplikacji internetowych. Połączenie dalej oparte jest o *TCP* jednak przesyłana wiadomość nie jest tekstowa jak w przypadku pierwszej wersji protokołu, a binarna. Szereg wad wcześniejszej wersji został usunięty przez dodanie nowych funkcjonalności. *HTTP/2* jest multipleksowany, przez co pobierając klika zasobów możemy robić to za użyciem pojedynczego połączenia. Dodatkowo zasobom można nadać priorytet. Usprawnieniem było także dodanie *Pipeliningu,* polegające na skanowaniu wymaganych zasobów i zaciąganie ich zbiorczo.
- *HTTP/3* – najnowsza wersja protokołu wydana w 2022. Nadal nie została w pełni zaadaptowana przez większość stron internetowych. Aktualizacja protokołu http przyniosła ze sobą znaczące zmiany, ponieważ ten nie jest już oparty o *TCP*, a o stratny protokół *UDP*. Głównym celem nowej wersji protokołu było usprawnienie możliwości protokołu *HTTP/2*. Bardzo dużą zaletą *HTTP/3* jest integracja z *TLS* (protokołu używanego do szyfrowania *http*). Dzięki temu przeglądarka nie musi wysyłać dodatkowych żądań w celu nawiązania połączenia *https*. Niestety protokół nadal nie jest używany przez większość stron oraz nie jest wspierany przez większość popularnych bibliotek.

Mówiąc o protokole *http* trzeba także wspomnieć o architekturze *REST* (*Representational state transfer*) [23]. *REST* posiada kilka zasad, które opisują, jak powinno używać się protokołu *http* tworząc serwer aplikacji:

- Korzystanie z architektury klient-serwer – opiera się na oddzieleniu interfejsu klienta aplikacji od implementacji jej logiki. Jest to podstawowa architektura, z której korzystają wszystkie aplikacje internetowe
- Bezstanowość – sama aplikacja nie powinna trzymać danych pomiędzy obsługą żądań użytkownika. Zasada ta nie dotyczy danych zapisywanych do bazy danych, ale mówi o niezapisywaniu danych w sesji użytkownika.
- *Cacheability* – zasób, do którego odwołuje się klient powinien być cechowany jako odpowiedni do trzymania w pamięci podręcznej lub nie. Dzięki takiemu zabiegowi klienci lub pośrednicy mogą zachować niektóre zasoby w swojej pamięci, tym samym zmniejszając ilość żądań.
- System warstwowy – serwer aplikacji może być poprzedzony pośrednikami. Dodanie dodatkowych pośredników nie powinno wpływać na działanie aplikacji
- Jednolite interfejsy – interfejs aplikacji powinien być odpowiednio opisany. Każdy zasób powinien być zidentyfikowany, jego reprezentacja wystarczająca do aktualizacji oraz posiadać minimalną ilość informacji do wykonania na nim operacji
- Kod na żądanie (Opcjonalne) – serwer potrafi wysłać kod rozszerzający lub dostosowujący funkcjonalność klienta

4.2. GraphQL

GraphQL [24] jest otwarto-źródłowym deklaratywnym językiem zapytań. Technologia przyjmuje styl komunikacji synchronicznej blokującej. Umożliwia on dostosowanie danych zwracanych przez serwer. Technologia zamiast udostępniać dane poprzez kilka punktów końcowych robi to używając tylko jednego. *GraphQL* potrafi złączyć różne udostępniane zasoby niezależnie od bazy danych. Poza pobieraniem danych technologia pozawala na wykonanie operacji na danych oraz subskrypcje zmian.

Ważnym elementem *GraphQL* jest definicja schematów (z angielskiego *scheme definition*). Serwer chcący udostępnić dane używając *GraphQL* musi określić model danych oraz ich typy. Schematy są tworzone przy użyciu specjalnego języka. Schemat posiada element *Query* oraz *Mutation,* które agregują zapytania obsługiwane przez serwer. Reszta schematów definiuje typy obiektów i ich pola. Domyślnie wszystkie pola w *GraphQL* są opcjonalne, ale mamy możliwość ich wymagania:

Listing 5. Przykład definicji schematu

```
extend type Query {
feed(profileId: UUID!): [PostProjection]!
}

type PostProjection {
postId: UUID
author: UUID
text: String
attachments: [AttachmentProjection]
sentTime: LocalDateTime
comments: [CommentProjection]
}
```

Klient chcący wysłać zapytanie do serwera musi opisać je za pomocą specjalnego języka zapytań. Zapytanie tworzymy poprzez określenie jakiego jest ono typu (*Querry* lub *Mutation)* i wyboru dostępnej operacji. Jeżeli jest to typ *Query* musimy jeszcze podać dane które chcemy uzyskać:

Listing 6. Definicja zapytania do pobrania danych

```
query {
fetchGroupPost(groupId:”01d9ae5c-bc86-43fc-8e16-0d144e2828a4”) {
postId
author
text
attachments {
attachmentId
resourceLink
type
    }
sentTime
  }
}
```

Listing 7. Definicja zapytania do edycji danych

```
mutation {
addGroupPost(post: {
group: „01d9ae5c-bc86-43fc-8e16-0d144e2828a4”
author: „30c90205-5f18-4e3a-8e3c-40fda63038ee”
text: „The greatest truth is there isn’t one”
attachments: [
      {
resourceLink: „http://youfly.com/123”,
type: „IMAGE”
      }
    ]
  })}
```

Twórcy *GraphQL* nie podają dokładnego opisu technicznego twierdząc, że jest to jedynie specyfikacja języka, która może bazować na dowolnym protokole. Mimo zapewnień autorów większość implementacji (w tym wszystkie dla języka *Java*) są stworzone na bazie *http*. Klient wysyła do serwera żądanie *http* o metodzie *POST* i typie *appliocation/graphql*.

Działanie protokołu jest skomplikowane. Klient przygotowuje *Query*, które jest tłumaczone na obiekt *JSON* wpisywany do ciała żądania *http*. Powstały obiekt *http* nosi specjalną nazwę *AST* (*Abstract Syntax Tree*). *AST* jest strukturą danych reprezentującą drzewo, gdzie węzły opisują atrybuty naszego zapytania. Powstałe drzewo posiada bardzo dużą ilość informacji i jest dużych rozmiarów nawet dla małego zapytania. Przykładowo zapytanie z „Listing 6” po przetworzeniu ma długość aż 224 linii kodu *JSON*. Żądanie po przyjęciu przez serwer jest przetwarzane w specjalnym cyklu składającym się z kilku faz:

- Asercja (*Assertion Process*) – wykonywana jest walidacja przesłanego schematu. Każdy element drzewa *AST* podlega walidacji. Sprawdzane jest czy schemat pasuje do zapytania oraz czy typy są zgodne

- Budowanie kontekstu (*Building the Execution Context*) – aby obsłużyć zapytanie silnik *GraphQL* musi posiadać szereg informacji jak np. typy opisane w schematach
- Wykonanie operacji (*Operation Execution*) – na tym etapie *GraphQL* przygotowuje nowe *AST* scalające informacje o operacji, którą otrzymał oraz schematach danych
- Wykonanie pól (*Field Execution*) – jest to główna faza procesowania zapytania *GraphQL*. Pola otrzymane w *AST* są analizowane i na ich podstawie podejmowana jest decyzja o pobraniu danych oznaczonych przez pole
- *Field Resolution* – operacja polega na wykonaniu *resolver'ów,* czyli operacji pobierających dane opisywane przez pola wskazane w żądaniu
- Dopełnienie Wartości (*Value Completion*) – po pobraniu danych sprawdzane są rekursywne zależności do innych schematów. Jeżeli w zapytaniu występują zagnieżdżenia do różnych obiektów, to dla każdego z tych obiektów wykonujemy rekursywnie osobny cykl rozpoczynając od *Operation Execution*

4.3. Broker wiadomości *Kafka*

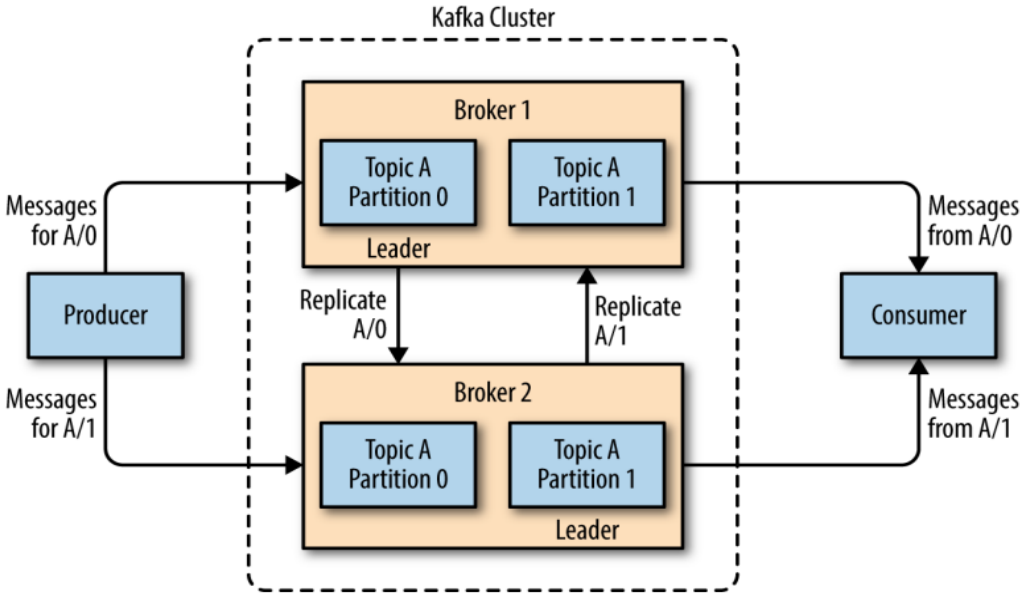
Apache Kafka [25] to narzędzie dostarczające szereg rozwiązań takich jak broker wiadomości, rozproszony magazyn zdarzeń oraz platforma przetwarzania strumieniowego. Broker wiadomości umożliwia wymianę komunikacji w stylu asynchronicznym. Komunikacja z brokerem działa w oparciu o binarny protokół *TCP,* jednak istnieją rozszerzenia pozwalające na komunikowanie się z użyciem innych protokołów dzięki *Kafka Connectors*.

Narzędzie wyróżnia się tym, że do działania potrzebuje uruchomienia kilku komponentów. *Kafka* do działania potrzebuje swojego klastra, czyli zbioru instancji aplikacji zarządzanych przez scentralizowany podmiot. Funkcję podmiotu przyjmuje *Zookeeper* [26], czyli narzędzie pozwalające na konfiguracje, synchronizacje oraz zarządzanie grupą procesów działających w rama jednego systemu rozproszonego. Podstawową jednostką w klastrze *Kafki* jest broker wiadomości. W obrębie jednego klastra może istnieć wiele instancji brokera, gdzie każda z nich obsługuje klientów przypisanych do niej.

Kafka wprowadza kilka definicji, które opisują jej poszczególne elementy:

- Wiadomość (*Message*) – najmniejsza jednostka danych, będąca pojedynczą informacją wysłaną przez klienta. Technicznie jest to tablica binarna zapisywana przez proces do zbioru logów oraz transportowana przez broker. Oprócz danych wiadomość może także zabierać klucz
- *Producent* – klienci działający w oparciu o broker wiadomości posiadają dwa typy. Jednym z nich są producenci, którzy odpowiadają za przesłanie wiadomości od instancji brokera
- *Consumer* – jest to drugi rodzaj klienta, który nasłuchuje na wiadomość przekierowaną do niego przez broker wiadomości
- • *Topic* – unikatowa nazwa, która oznacza kanał wiadomości do którego możemy wysłać wiadomość lub jej nasłuchiwać
- *Commit Log* – jest to baza danych wiadomości, które odebrał broker wiadomości. Po odebraniu wiadomości od *Producent*'a, ta jest najpierw zapisana do *Commit Log*, a następnie wysłana do klientów nasłuchujących
- Partycja (*Partition*) – jest to część zbioru wiadomości z określonego *Topic'a* przypisana konkretnej instancji brokera. Klient nasłuchujący może korzystać tylko z jednej partycji jednocześnie
- *Batch* – klient publikujący wiadomość dla jednego partycji *topic'a* potrafi zebrać kilka kolejnych wiadomości i wysłać je zbiorczo. Proces ten bezpośrednio wpływa na wydajność transportu informacji i jest konfigurowalny przez programistę.

- *Serializer* – dane przekazywane przez *Producent'a* do brokera są najpierw konwertowane do bajtów
- Deselializer – otrzymane dane są w pierwszej kolejności konwertowane przez Consumer'a do swojej pierwotnej formy



Rysunek 3. Poglądowy schemat działania brokera wiadomości *Kafka* [27]

Większość języków programowania posiada implementacje biblioteki pozwalającej w łatwy sposób zintegrować się z brokerem. Obowiązkiem programisty jest jedynie skonfigurowanie *Procudent'a* lub *Consumera*

Listing 8 Przykładowych kod wysyłający wiadomość do brokera przy użyciu języka *Java*

```
Properties props = new Properties();
props.put(„bootstrap.servers”, „localhost:9092”);
props.put(„value.serializer”,
„org.apache.kafka.common.serialization.StringSerializer”);
props.put(„key.serializer”,
„org.apache.kafka.common.serialization.StringSerializer”);
Producer<String, String> producer = new KafkaProducer<>(props);
producer.send(new ProducerRecord<String, String>(„my-topic”, „This is message”);
producer.close();
```

Listing 9 Przykładowo kod *Java* nasłuchujący wiadomości przysłane pod wskazany *topic*

```
Properties props = new Properties();
props.setProperty(„bootstrap.servers”, „localhost:9092”);
props.setProperty(„key.deserializer”,
„org.apache.kafka.common.serialization.StringDeserializer”);
```

```
props.setProperty(„value.deserializer”,
„org.apache.kafka.common.serialization.StringDeserializer”);
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList(„my-topic”));
while (true) {
ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(1000));
System.out.printf(„Message value: %s”, record.value());

 }
```

4.4. Zdalne wywołanie procedury *grpc*

Grpc [28] to narzędzie pozwalające na zastosowanie wzorca zdalnie wykonanej procedury (*Remote Procedure Calls*). Nazwa wzięła się od twórców technologii, czyli firmy *Google*. Technologia powstała w celu umożliwienie wydajnej komunikacji pomiędzy wieloma mikrousługami. Protokół jest zbudowany na bazie *HTTP/2*. Technologia, mimo że została zaprojektowana na potrzeby architektury mikrousług jest wykorzystywana w wielu innych typach systemów.

Dodatkowym usprawnieniem jest wykorzystanie *ProtoBuff*, która pozwala na określenie schematu wiadomości z którego następnie generowane są *serializatory* i *deseliarizatory* (służące na konwersje obiektów na wiadomości). Taki zabieg pozwolił na odcięcie technologii od konkretnego języka programowania oraz na generowanie kodu na potrzeby wymagań projektowych:

Listing 10. Przykład schematu, z którego generowana jest projekcja wiadomości

```
message LoadPostsWithGroupPostsMessage {
string session = 1;
repeated string profiles = 2;
repeated string groups = 3;
int32 size = 4;
int32 page = 5;

}
```

Listing 11. Przykład schematu definiującego klienta aplikacji oraz kod służący do implementowania serwera

```
service  GroupService {
rpc loadGroup (LoadGroupMessage) returns (LoadGroupResponse);
rpc loadGroupPosts (LoadGroupPostsMessage) returns (LoadGroupPostsResponse);
rpc loadPosts (LoadPostsMessage) returns (LoadPostsResponse);
}
```

Po określeniu schematu dla serwisu oraz wiadomości programista ma obowiązek uruchomić proces generujący biblioteki dla wskazanych przez siebie języków programowania. Rezultatem wykonanej operacji są projekcje (w zależności są to gotowe klasy lub struktury). Poza projekcjami wygenerowana biblioteka zawiera także klienta dla serwisu, który jest gotowy do użycia. Najważniejszym elementem jest wygenerowana klasa, którą musi rozszerzyć serwer w celu obsługi żądań.

5. Analiza porównawcza

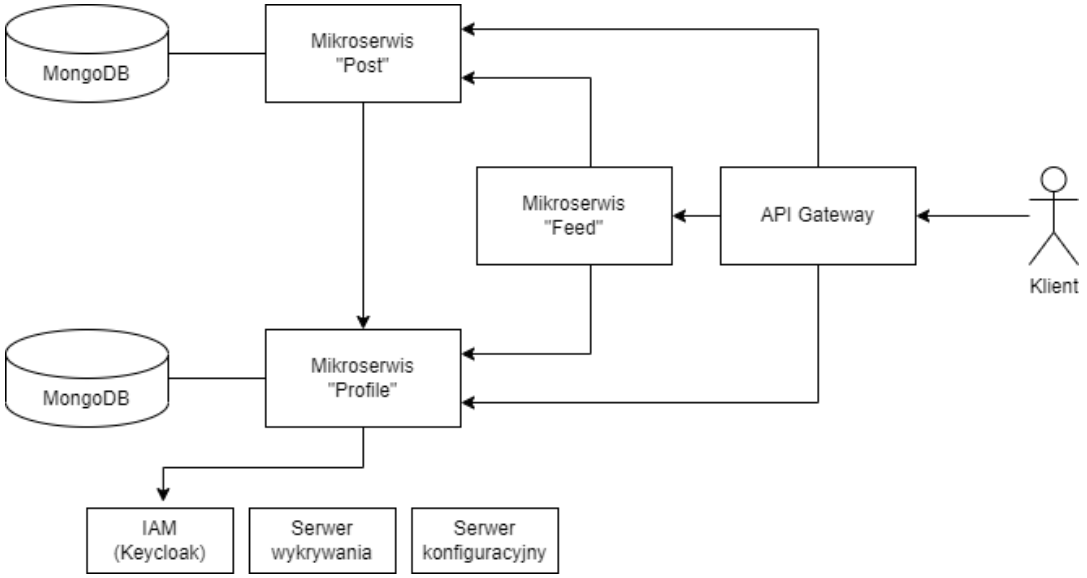
Porównanie technologii zostało sporządzone poprzez uwzględnienie kilku różnych perspektyw. Założenie jednego punktu widzenia wpływa negatywnie na ocenę, ponieważ nie odzwierciedla ona odczuć różnych specjalistów współpracujących z daną technologią. Protokół, który jest lubiany przez programistów ze względu na swoją prostotę może okazać się niewydajny. Technologia, którą proponuje architekt oprogramowania, może wymagać dodatkowych zasobów, których docelowy odbiorca nie potrafi zapewnić. Aby rzetelnie ocenić każdą z technologii uwzględniono kilka perspektyw:

- Architektura – każda z technologii wpływa w mniejszym lub większym stopniu na schemat architektury naszej aplikacji. Podczas projektowania systemu wybór wzorca komunikacji wpływa znacząco na efekt końcowy. Dodatkowo niektóre technologie, choć rozwiązują pewne problemy, potrafią wprowadzić więcej złożoności
- Programista – najbardziej detaliczną pracę w procesie tworzenia oprogramowania wykonuje programista. Punkt widzenia osoby piszącej kod jest kompletnie inny niż osoby operującej na wyższym poziomie abstrakcji. Najważniejszą cechą technologii dla przeciętnego programisty jest jej prostota i zwięzłość rozwiązania. Poza tym niektóre technologie wywierają na programiście użycie określonego paradygmatu programowania
- Klient – API wystawione przez system oparty o mikroserwisy ma swoich odbiorców, którzy muszą podporządkować się standardom jakich używa nasza aplikacja. W zależności od użytej przez nas technologii implementacja aplikacji klienckiej będzie mniej lub bardziej skomplikowana, ograniczała lub dawała dodatkowe możliwości
- Wydajność – główną metryką w stosunku do poszczególnych technologii jest jej wydajność. Protokoły komunikacyjne mogą różnić się pod wieloma względami. Poszczególne cechy brane pod uwagę przy ocenie wydajności to:
    - Końcowa latencja
    - Czas blokowania klienta
    - Wielkość wiadomości
    - Używana infrastruktura

5.1. Technologie z perspektywy architektury

Ocena technologii z perspektywy architektury jest efektem projektowania aplikacji pod kątem użycia danej technologii. Projektując aplikacją zwrócono uwagę na praktyki oraz wzorce wykorzystywane podczas projektowania systemu opartego o dany wzorzec komunikacji. Ocenie podlegał wpływ technologii na architekturę, swoboda w jej projektowaniu, wprowadzenie dodatkowej złożoności, rozwiązane problemu, potencjalne problemy i ograniczenia technologii.

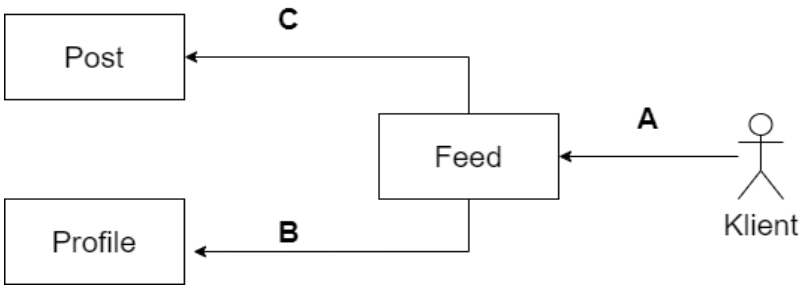
5.1.1 HTTP



Rysunek 4. Schemat aplikacji przy użyciu protokołu *http*

Protokół z racji, że używa prostego modelu komunikacji, czyli żądanie i odpowiedź, nie wprowadza złożoności do architektury. Jednak w celu poprawnego działania aplikacji należy wprowadzić dodatkowy mikroserwis, który będzie odpowiadał za wykrywanie nowych usług (z angielskiego *discovery service*). Przy projektowaniu serwera zależy nam na umożliwieniu łatwego skonfigurowania go, czyli bez potrzeby manualnego wprowadzania adresu sieciowego dla każdej usługi. Mimo, że w przypadku omawianej aplikacji istnieje zaledwie kilka usług to już średniej wielkości aplikacje mogą składać się z kilkudziesięciu. Ustawianie adresów ręcznie lub za pomocą skryptu jest uciążliwe i podatne na błędy, z tego względu architektura używająca protokołu *http* musi zawierać *discovery service.* Kolejnym problemem stawianym przez ten protokół jest tak zwany *load balancing,* czyli rozkładanie nadchodzących zapytań na kilka dostępnych instancji serwera docelowego, aby nie obciążać pojedynczej instancji. Zwykle rozwiązaniem tego problemu jest implementacja rozkładania zapytań na dostępne instancje podczas wysyłania zapytań przez klienta lub usługę. Powyższe problemy są na tyle powszechne, że większość języków oprogramowania posiada swoją implementację klienta umożliwiającego *load balancing* i komunikacje z *discovery server*. Wystawienie dodatkowego serwisu nadzorującego naszą aplikację także nie wymaga dużych zasobów. Podczas pracy z implementacją *discovery server* czyli *Spring Eureka Server* ta nie przekroczyła 100MB pamięci RAM oraz 5% zużycia procesora. Jedynym minusem *discovery server* jest dodanie złożoności wdrażania aplikacji oraz wytworzenie kilku jej replik, ponieważ gdy zabraknie *discovery server* nowe usługi nie będą mogły komunikować się z pozostałymi.

Natomiast natura protokołu *http* ma jedną zasadniczą wadę z punktu widzenia architektury, a jest nią tak zwany *temporar coupling* [29]. Problem ten polega na sztywnym chwilowym powiązaniu mikrousług. Projektując tę architekturę staramy się ograniczyć wszelkie powiązania do minimum. Powiązania pomiędzy mikrosuługami wprowadzają dodatkowe problemy i tak jest w tym przypadku. Załóżmy poniższy scenariusz:



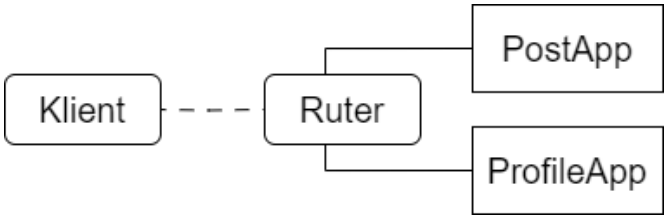
Rysunek 5. Przykładowy schemat wykonania zapytania obrazujący problem protokołu *http*

Zapytania wykonywane są w następującej kolejności: A (zapytanie klienta), B (pobranie informacji o obserwowanych profilach i grupach), C (pobranie informacji o postach). Problemem staje się tak zwany *timeout,* czyli czas oczekiwania klienta *http* na odpowiedź od serwisu. Dodatkowo w celu zapewnienia dodatkowej odporności stosuje się próby ponownego wysłania wiadomości oraz opcjonalne wstrzymanie komunikacji, jeżeli ten jest nieosiągalny. Przykładowo ustawiając *timeout* na 100 milisekund w obrębie mikrousług wartość ta będzie 3 krotnie większa dla klienta w naszym przypadku. Jeżeli stosujemy wspomniane wcześniej próby oraz wstrzymanie komunikacji, to czas ten rośnie wykładniczo. Jednak jeszcze większy problem pojawia się, kiedy jedna z usług np. B jest chwilowo nieosiągalna. W takim przypadku użytkownik jest pozbawiony funkcjonalności udostępnionej przez serwis *Feed,* tym samym pozbawiamy się istotnej zalety architektury mikrousług. Problem jest bardziej krytyczny, gdy komunikacja pomiędzy usługami ma formę zdarzeń, czyli usługa emituje jedynie zdarzenie do drugiej o wykonaniu jakiejś operacji. Komunikacja taka jest najczęściej czysto techniczna i nie jest ona znana klientowi aplikacji. Byłoby dużym błędem gdybyśmy uzależniali działanie całej aplikacji od dostępności wszystkich mikrousług w tym samym czasie. Problem ten można naprawić poprzez dostarczenie dodatkowej implementacji, która odkładałaby zdarzenia do bazy danych w chwili nieobecności którejś z usług, po czym propagowała je w określonych odstępach czasu.

5.1.2 Graphql

Architektura w przypadku *graphql* wygląda podobnie w porównaniu do *http*. Jednak *graphql* nie został wykorzystany przy komunikacji wewnętrznej. Powodem takiej decyzji jest brak zalet protokołu w przypadku takiego użycia. *Graphql* został zbudowany docelowo dla klientów systemu, którzy dzięki tej technologii dostali możliwość szczegółowego określenia wymaganych przez nich zasobów. Możliwość określenia potrzebnych zasobów nie jest przydatny przy komunikacji pomiędzy usługami, które mają sztywno określony kontrakt. Dodatkowo udostępniony przez mikrousługę interfejs powinien być mały zgodnie z koncepcją architektury.

Chociaż protokół jest zaprojektowany dla aplikacji korzystającej z systemu, to twórcy *graphql* przygotowali koncepcję przydatną z punktu widzenia architektury systemu jaką jest federacja. Federacja jest rozszerzeniem wzorca kompozycji API, która jest szeroko stosowana w mikrousługach. Polega ona na ustawieniu usługi pośredniczącej pomiędzy klientem a usługami docelowymi. W przypadku *http* chcąc nie zmuszać użytkownika do wysyłania osobnych żądań do różnych mikroserwisów implementujemy *apigateway,* który mapuje zapytania do serwisu docelowego. Chcąc nie zmuszać użytkownika do wysyłania szeregu wiadomości często poszerza się obowiązki *apigateway,* który potrafi rozdzielić jedno zapytanie na kilka pomniejszych. Takie rozwiązanie jest pracochłonne. Natomiast federacja *graphql* dostarcza takiego rozwiązania automatycznie.



Rysunek 6. Przykładowy schemat federacji

Projektując pojedynczą usługę, *graphql* wymusza zadeklarowanie zasobów, które są znane dla rutera. Dzięki temu jest on w stanie automatycznie mapować oraz rozdzielać przychodzące zapytania, bez narzutu pracy związanego z implementowaniem pośrednika. W przypadku dużych, które udostępniają setki różnych zasobów takie rozwiązanie potrafi ukryć złożoność systemu. Rozwiązanie federacji jest szeroko wykorzystywane w takich aplikacjach jak *Netflix* oraz *Facebook,* które utrzymują własne implementacje *graphql* jako otwarto-źródłowe biblioteki.

Niestety w chwili pisania pracy twórcy *graphql* podają, że nie wszystkie technologie są w pełni kompatybilne z ruterem federacji. Problem nie dotyczy technologii wykorzystanych do stworzenia omawianej aplikacji. Natomiast federacja posiada ograniczenie jakim jest integracja z elementami systemu, które nie korzystają z *graphql.* Istnieje rozwiązanie, które na podstawie oferowanego *REST API* jest w stanie obsłużyć zapytania *graphql*, jednak jest ono dostępne jedynie dla języka *JavaScript.* Drugim powszechnym rozwiązaniem tego problemu jest dostarczenie własnej implementacji rutera lub rozszerzenie istniejącej, o ile ta daje taką możliwość.

5.1.3 Grpc

Architektura powstała przy uwzględnieniu *grpc* jest identyczna do tej powstałej z użyciem *http,* co wynika z faktu, że protokoły te opierają się na komunikacji żądanie-odpowiedź. Protokół jest bardziej przystosowany do komunikacji pomiędzy mikrousługami i elementami, które są „świadome" infrastruktury sytemu. Udostępnienie *grpc* klientom *API* może odbyć się na 2 sposoby, udostępniając infrastrukturę usług lub stworzyć *API* pośredniczące w celu ukrycia złożoności systemu. Pierwsze rozwiązanie jest odpowiednie, kiedy usług jest kilka lub użytkownik końcowy potrzebuje jedynie kilku z nich. W takim wypadku można udostępnić usługi oraz podać ich adresy sieciowe dla klienta. Jest to dość problematyczne do zorganizowania, jeśli chodzi o konfiguracje. Drugie rozwiązanie wydaje się być niemożliwie lub bardzo pracochłonne w dużych systemach. Zasoby udostępnione przez np. 50 usług muszą być w tym przypadku uwzględnione w nowym serwisie pośredniczącym. *Grpc* nie posiada rozwiązań, które pozwalałyby w deklaratywny sposób przekierować dany zasób do odpowiednich usług, więc rozwiązanie takie należy zbudować od zera.

Ważnym elementem *grpc* jest możliwość utrzymania połączenia *tcp* pomiędzy zapytaniami. Czas tego połączenia może być dostosowany przez programistę. Z punktu widzenia optymalizacji czas połączenia powinien być jak najdłuższy, ponieważ pomijamy w ten sposób ustawianie połączenia pomiędzy dwoma adresami. Długi czas utrzymania połączenia nie jest rekomendowany w przypadku mikrousług. W architekturze tej chcemy ograniczyć powiązania pomiędzy usługami pod względem ilości jak i czasu. Wydłużając czas utrzymania połączenia zmniejszamy skalowalność systemu. *Grpc* przy łączeniu się ze zbiorem usług korzysta z *load balancing'u.* Utrzymanie długiego połączenia odbiera możliwość przekierowania ustawionego już połączenia do innej instancji usługi. Może być to problematyczne przy nagłym wzroście ruchu sieciowego. Usługa połączona już z jedną instancją nie przekieruje się do innego przez co obciąży tylko jedną instancję.

Wydłużanie czasu połączenia jak i strumieniowanie danych umożliwione przez *grpc* jest dużą zaletą przy systemach, których elementem są urządzenia pobierające i wysyłające dane w czasie rzeczywistym. Ustawienie długiego czasu utrzymania połączenia do specjalnie przygotowanej usługi potrafi odciążyć urządzenia, które nie posiadają dużych zasobów obliczeniowych. Dodatkowo

utrzymanie połączenia znacznie zmniejsza latencję, co jest atrakcyjne z punktu widzenia systemów, które przetwarzają dane pochodzące z sensorów w czasie rzeczywistym.

5.1.4 Kafka

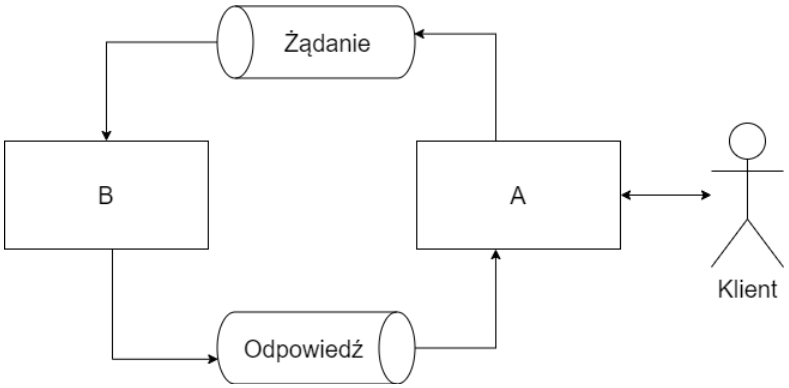
Kafka jest najbardziej złożoną technologią, która bardzo komplikuje architekturę. Powodem komplikacji jest wprowadzenie innego modelu komunikacji. Kafka oparta jest na przesyłaniu zdarzeń i procesowaniu ich w sposób asynchroniczny. Oznacza to, że usługi są od siebie całkowicie niezależne. Serwis chcący wysłać wiadomość nie musi oczekiwać, że inne serwery w systemie będą działać. Kafka działa jako pośrednik odbierając wiadomość, zapisując ją w pamięci i wysyłając ją do odbiorców nasłuchujących na nową wiadomość. Serwis wysyłający wiadomość nie musi czekać na przetworzenie wiadomości. Również, jeżeli odbiorca wiadomości będzie obciążony lub nieosiągalny ta zostanie zapisana w kolejce i przeprocesowana, kiedy będzie taka możliwość.

Oparcie komunikacji na architekturze zdarzeń pozwala na implementowanie wielu rozwiązań, które nie są możliwe stosując inny model komunikacji. Kilka rozwiązań, które opierają się na takim modelu zdarzeń to *CQRS (Command Querry Responsibility Segregation)*, transakcje rozproszone (choreografia *SAGAS*), *Event Sourcing* z wykorzystaniem *Kafki* jako bazy danych, kolejka priorytetowa, *politness* i przetwarzanie strumieniowe.

Kafka stawia kilka wyzwań podczas projektowania aplikacji. Inaczej niż w przypadku pozostałych technologii mikrousługni nie są powiązane ze sobą, a są powiązane z klastrem kafki. Inaczej są również zaprojektowane operacje wykonywane z użyciem kilku usług. Dzielą się one na dwa typy. Pierwszym są wysyłane zdarzenia, które informują o wykonaniu jakiejś operacji. Przykładowo użytkownik, który chce zostać dodany do grupy wysyła żądanie do mikroserwisu *Post,* który obsługuje zawartość grup. Następnie informacja o dodaniu użytkownika do grupy jest emitowana. Mirkousługa *Post* musi jedynie wyemitować zdarzenie nie biorąc żadnej odpowiedzialności za odczytanie jej przez odbiorców. Skutkuje to dwoma efektami. Pierwszym efektem jest brak chwilowego sprzężenia pomiędzy usługami. Natomiast wprowadza to tak zwane *eventual consistency,* które oznacza czas niespójności danych. Niespójnością danych w przypadku omawianej aplikacji będzie czas potrzebny do przetworzenia przez system informacji wyemitowanej przez usługę *Post.* Drugim typem wiadomości są komendy, które różnią się intencją wysłania. Usługa wysyłając komendę oczekuje efektu, który może być odpowiedzią jak i zdarzeniem. Biorąc pod uwagę wewnętrzne operacje systemu możliwe jest luźne wykonanie operacji w którym zleceniodawca po pewnym czasie otrzymuje zdarzenie o potwierdzające poprawne wykonanie lub zleca je podobne. Sytuacja komplikuje się, kiedy operacja zostaje zlecona przez użytkownika systemu. Użytkownik zwykle chce znać efekt wykonania operacji w formie odpowiedzi otrzymanej tuż po wysłaniu wiadomości. Problemem są luźne połączenia systemu, użytkownik po wysłaniu wiadomości protokołem opartym o żądanie-odpowiedź może dowiedzieć się jedynie o przyjęciu wiadomości do systemu. Niektóre operacje takie jak np. obserwowanie znajomego lub usunięcie grupy mogą być zaaranżowane w ten sposób. Jednak, kiedy użytkownik chce pobrać listę nowych postów wymagane jest przez mapowanie asynchronicznego sposobu komunikacji na synchroniczny. Można to wykonać na dwa sposoby.

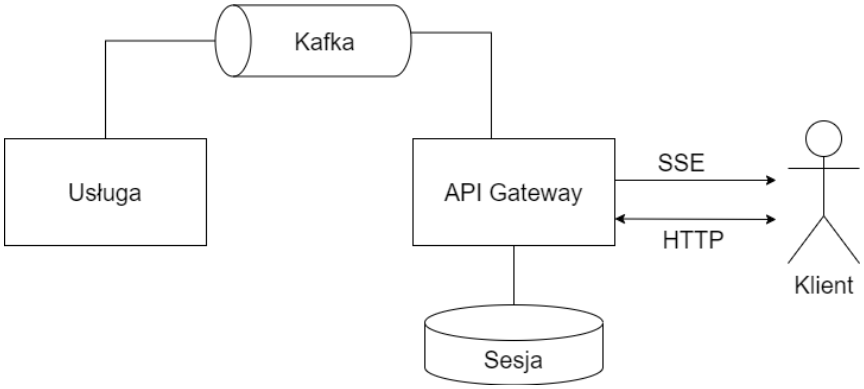
Istnieje wzorzec architektoniczny, który umożliwia przekształcenie komunikacji. Przykładowo istnieją serwisy A oraz B. Serwis A otrzymuje żądanie od użytkownika, który chce pobrać listę grup. Serwis A wysyła żądanie w formie wiadomości do kolejki z dodatkowym oznaczeniem (identyfikatorem korelacji). Następnie serwis B odbiera wiadomość i po wykonaniu zadania zwraca odpowiedź do drugiej kolejki. Wątek serwisu A jest zablokowany (blokuje także klient) do momentu, w którym odczyta wiadomość oznaczoną odpowiednim identyfikatorem. Po odebraniu odpowiedniej wiadomości zawraca ją klientowi. Takie rozwiązanie problemu jest wspierane przez *kafkę*. Posiada ono jednak kilka problemów. Podczas zablokowania klienta trzeba wyznaczyć *timeout* po którym odblokujemy go z informacją o niepowodzeniu. Nieodblokowanie klienta skutkowałoby dużym obciążeniem systemu, który posiada ograniczoną liczbę wątków. Poza *timeoutem* narzuconym przez serwer istnieje także ten który ustalił klient łączący się z systemem. W wyniku luźnego klient

poinformowany o wygaśnięciu połączenia z serwerem nie będzie poinformowany o efekcie operacji, którą wysłał a które została przetworzona w oddzielnym serwisie. Projektując system z takim rozwiązaniem należy wyznaczyć czas odpowiedzi systemu i skalować go w taki sposób, aby uniknąć sytuacji, w której klienci nie będą poinformowanie o efekcie operacji.



Rysunek 7. Schemat przekształcenia wiadomości asynchronicznej na synchroniczną

Drugim sposobem na rozwiązanie problemu jest utworzenie strumienia danych pomiędzy klientem a serwerem. Połączenie takie może być umożliwione przez użycie dodatkowych protokołów takich jak *Websocket, Rsocket* lub *Server Sent Events* (która jest integralną częścią protokołu *http*). Poza użyciem dodatkowego protokołu wymuszamy na kliencie utworzenie dwóch połączeń. Pierwszym połączeniem jest zlecenie operacji, które nie różni się od poprzednich implementacji. Drugim połączeniem, które musi nawiązać klient jest zasubskrybowanie strumienia informacji udostępnianej przez serwer w czasie rzeczywistym. Poza zmianami od strony klienta wymagane są dodatkowe rozwiązania w komunikacji wewnętrznej systemu. Klient, który wywołuje dwa osobne żądania musi mieć przypisany unikalny identyfikator tak aby usługa odbierająca żądanie mogła oznaczyć strumień danych wychodzący do klienta. Identyfikator musi być także przekazany klientowi, aby ten mógł go okazać podczas tworzenia strumienia danych (tak zwany identyfikator sesji użytkownika). Serwer, aby przekazać identyfikator może użyć *cookies* które są mechanizmem udostępnionym przez *http*. Udzielony identyfikator musi być także zapamiętany przez serwer w celu skojarzenia użytkownika z otwartym strumieniem. Serwer, który trzyma połączenie z klientem nasłuchuje w osobnym wątku na wynik wcześniej przekazanej operacji. Kiedy wynik pojawi się zostaje on przekierowany do klienta poprzez wcześniej otwarty strumień informacji. Rozwiązanie komplikuje się jeszcze bardziej kiedy chcemy skalować serwis odpowiedzialny za zarządzanie strumieniem użytkownika. Kiedy istnieje kilka instancji serwisu zarządzającego połączeniem z klientem wymagane jest użycie dodatkowej bazy danych która będzie udostępniała informacje na temat instancji, na której obecnie znajduje się obecnie połączenie oznaczone identyfikatorem użytkownika. Odczytywanie odpowiedzi może być wykonywane przez wszystkie instancje jednocześnie i odrzucane, jeżeli nie są w posiadaniu wskazanego połączenia. Odczytywanie wiadomości i odrzucanie nie są optymalnym rozwiązaniem, dlatego należy wprowadzić mechanizm przekierowujący odczytaną wiadomość do odpowiedniej instancji, może być to osobny mikroserwis który zarządza sesją użytkownika lub zaimplementowanie nowej funkcjonalności w serwisie zarządzającym połączeniem. W opisanej aplikacji przyjęto rozwiązanie ze strumieniem informacji, ale zrezygnowano z implementacji dodatkowego serwisu zarządzającego sesją (jeżeli zła instancja odbierze wiadomość ta zostaje odrzucona).



Rysunek 8. Alternatywne rozwiązanie komunikacji między klientem a systemem opartym o zdarzenia

Użycie *kafki* zmienia także infrastrukturę projektu. Mikrousługi nie muszą wiedzieć o swoim istnieniu, dlatego nie potrzebny jest *doscovery service.* Natomiast kafka jest technologią, która do działania potrzebuje klastra. Klaster składa się z dwóch elementów: *zookeeper* oraz broker kafki. Ilość brokerów jest konfigurowalna i zależy od skalowania aplikacji. Klaster kafki potrzebuje minimum 6GB pamięci operacyjnej, aby działać poprawnie, procesora o 4 rdzeniach oraz zasób pamięci adekwatny do ilości danych przesyłanych w systemie. Kafka jest najbardziej wymagająca pod względem infrastruktury z omawianych technologii.

Ważnym elementem brokera wiadomości jest zapisywanie przesyłanych wiadomości w lokalnej pamięci. Pozwala to na uzyskanie większej odporności systemu oraz na utrzymanie historii przesyłanych wiadomości co jest pomocne projektując system, który musi być wysoce odporny na błędy i awarie.

Kafka pozwala także na skonfigurowanie wielu elementów wpływających na jej działanie. Parametry takie jak *batch size* określający ilość wiadomości wysyłanych zbiorczo, czy *linger* określający czas bezczynności konsumenta wpływają na przepustowość i latencję końcową komunikacji. Obowiązkiem osób projektujących system jest dobranie takich parametrów dla każdego kanału komunikacji tak, aby odpowiadały one jego wymaganiom. Nie jest możliwe jednoczesne zapewnienie minimalnej latencji końcowej oraz maksymalnej przepustowości. Niektóre z ustawień są proporcjonalne do przepustowości, ale odwrotnie proporcjonalne do latencji końcowej.

Poza konfiguracją związaną z optymalizacją systemy *kafka* pozwala także dostosować funkcje związane z niezawodnością systemu. Producent po odebraniu wiadomości może oczekiwać odpowiedzi od brokera, odpowiedzi od wszystkich jego replik lub nie oczekiwać żadnej wiadomości. Istnieją także parametry określające sposób dostarczenia wiadomości. Producent *kafki* może być ustawiony jako idempotentny dzięki czemu będzie wysyłać jedną wiadomość bez powielania jej. Natomiast konsument ma dowolność, jeśli chodzi o określenie ilości prób odczytania wiadomości oraz ustawienia transakcji w czasie przetwarzania wiadomości. Każdy parametr zwiększający niezawodność systemu zmniejsza jego przepustowość lub latencję.

Kafka wprowadza wiele możliwości, których nie daję żadna z innych omawianych technologii. Historia wiadomości, możliwość oparcia komunikacji na zdarzeniach, pełna niezawodność, dostosowanie wydajności i niezawodność oraz autonomiczność usług są cechami, których nie da się uzyskać stosując inny model komunikacji. Jednak już podczas projektowania systemu *kafka* wprowadza bardzo dużo złożoności i problemów które były nieobecne w przypadku innych protokołów. Większość z wprowadzanych wyzwań wymaga zaimplementowania nowych mechanizmów w systemie. Dlatego przed użyciem *kafki* w systemie trzeba rozważyć inne alternatywy oraz oszacować czy wprowadzenie jej nie będzie zbyt pracochłonne. Wprowadzanie tak

skomplikowanego modelu komunikacji w mniejszych projektach może okazać się zbędne i generować koszty. Kolejnym minusem *kafki* jest brak optymalnego sposobu na przetworzenie żądań pobierających dane. Korzystanie z *kafki* do pobierania danych wiąże się z dodatkowym nakładem pracy i uzyskaniem gorszej wydajności niż w przypadku protokołów opartych na modelu żądanie odpowiedź.

5.2. Technologie z perspektywy programisty

Perspektywa programisty zawiera detaliczny opis w jaki sposób zintegrować daną technologię z systemem. Oceniono takie cechy technologii jak wsparcie wśród dostępnych języków, dostępne biblioteki, wymuszenie dodatkowej pracy na programiście lub użycie określonego paradygmatu programowania.

5.2.1 HTTP

Według statystyk *JetBrains* dotyczących mikroserwisów, aż 81% [22] ankietowanych twierdzi, że używa protokołu *http* do komunikacji. Powodem takiego stanu jest prostota w użytkowaniu protokołu. *Http* posiada wsparcie w każdej możliwej technologii, a sam protokół jest ustandaryzowany. Projektując system nie wprowadzamy dodatkowych wyzwań programiście, ponieważ komunikacja w oparciu o model żądanie- odpowiedź jest naturalna i nie wymaga asynchronicznego podejścia do tworzenia aplikacji.

Największą zaletą z punktu widzenia programisty jest mnogość otwarto-dostępnych bibliotek rozszerzających funkcjonalności protokołu. Część z nich jest tak zaawansowana, że z punktu widzenia programisty zewnętrzna usługa jest używana jak obiekt znajdujący się na tej samej maszynie. Przykładowo podczas tworzenia omawianej aplikacji wykorzystano bibliotekę *Spring Openfeign*. Zawiera ona szereg gotowych implementacji klienta *http*. Klienci są wysoko rozwinięci, ponieważ udostępniają możliwość konfiguracji funkcjonalności takich jak *load balancing*, *backoff (*wstrzymanie komunikacji*)*, *retry (*ponawianie próby komunikacji*)* oraz automatyczne integrowanie się z serwerem wykrywania. Podobne biblioteki są dostępne także w innych językach programowania.

Listing 12. Przykład użycia biblioteki do tworzenia klienta *http*

```
@FeignClient(„profile-app”)
interface ProfileServiceFeignClient {

@PostMapping(„/profile/{profileId}/group”)
fun addGroupToProfile(@PathVariable profileId: UUID,

@RequestParam(„groupId”) groupToAdd: UUID)
@DeleteMapping(„/profile/{profileId}/group”)
fun removeGroupFromProfile(@PathVariable profileId: UUID,

@RequestParam(„groupId”) groupToAdd: UUID)
@DeleteMapping(„/profile/removeGroupAssociations”)
fun removeGroupFromProfiles(@RequestBody data: RemoveGroupAssociations)
}
```

Klient *http* przedstawiony powyżej jest napisany w sposób deklaratywny. Użyta biblioteka na podstawie zadeklarowanego interfejsu dostarczy gotową implementację. Adnotacja *@FeignClient(„profile-app”)* mówi o tym, że interfejs zawiera opis klienta *http* oraz adres docelowy usługi (lub jak w tym przypadku nazwę usługi przypisaną w serwerze wykrywania). Pozostałe

adnotacje naniesione na metody *@PostMapping* oraz *@DeleteMapping*, wskazują na docelowe zasoby udostępnione przez podaną usługę.

Biorąc pod uwagę powyższe czynniki nie dziwi fakt, że protokół *http* jest powszechnie wykorzystywany oraz traktowany jest jako domyślny.

5.2.2 Graphql

Obecnie jedynie 14% [22] programistów deklaruje użycie *graphql*. *Graphql* posiada duże wsparcie w zakresie oferowanych bibliotek w najpopularniejszych językach. W chwili pisania pracy nie wszystkie funkcje oferowane przez *graphql* są dostępne w mniej popularnych językach (dotyczy to niektórych funkcji np. federacji). Protokół jest zdecydowanie mniej rozpowszechniony niż inne omawiane technologie jednak posiada wystarczająca ilość kontrybucji oraz bibliotek, aby używać go w środowisku komercyjnym.

Programista chcący wykorzystać *graphql* w swoim projekcie musi określić schemat zapytań, które mogą być wysłane do *API (*przykładowe schematy zostały umieszczone w „Listing 5”*).* Następnie można przystąpić do tworzenia samych punktów końcowych. Dla porównania *http* wymaga wprowadzenia do kodu koncepcji oraz dodatkowych technologii, aby udokumentować punkty końcowe, ustandaryzować linki oraz określić schemat zapytań. W przypadku *graphql* link wystawiony przez aplikację jest jeden. Schemat wraz z walidacją jest zadeklarowany w postaci schematu grafu. Jest to duża przewaga nad protokołem *http*, ponieważ narzucenie utworzenia schematu podczas tworzenia *API* jest bardziej komfortowe, niż trzymanie się koncepcji i ręczne dodawanie kolejnej warstwy z dokumentacją. Dodatkowo *graphql* umożliwia zadeklarowanie typów dzięki czemu programista nie musi dostarczać walidacji typów przekazanych w parametrach zapytania.

Tworzenie punktów końcowych nie różni się znacznie od tych stworzonych przy użyciu protokołu *http,* od strony kodu. W omawianej aplikacji wykorzystano bibliotekę Spring *Graphql* do implementacji usług. Poniżej umieszczono przykładową implementację serwera wykorzystującego *graphql.*

Listing 13. Przykładowy fragment kodu implementujący część serwera odpowiedzialnego za pobieranie informacji o grupach

```
@Controller
@CrossOrigin
class GroupsResolver(val postAppService: PostAppService, val profileAppService:
ProfileAppService) {

        @QueryMapping
        fun group(@Argument id: UUID, graphQLContext: GraphQLContext):
GroupProjectionGraphQl {
            val groupProjection = postAppService.getGroup(id,
graphQLContext.getOrDefault("Authorization", ""))
            graphQLContext.put("group", groupProjection)
            return GroupProjectionGraphQl(
                    groupInt = groupProjection.groupInt,
                    name = groupProjection.name,
                    description = groupProjection.description,
                    image = groupProjection.image,
                    posts = groupProjection.posts)
        }

        @SchemaMapping(field = "profiles", typeName = "GroupProjection")
        fun profiles(graphQLContext: GraphQLContext): List<ProfileProjectionQL>? {
```

```
            val groupProjection = graphQLContext.getOrDefault("group", null) as
GroupProjection?
            if(groupProjection != null) {
                return
profileAppService.getProfiles(groupProjection.profiles.joinToString(separator =
",") , graphQLContext.getOrDefault("Authorization", ""))
                    .map { profileProjection ->
                        ProfileProjectionQL(
                            profileID = profileProjection.profileID,
                            username = profileProjection.username,
                            birthday = profileProjection.birthday
                        )
                    }
            }
            return null
        }
}
```

Podobnie jak w przypadku *http* technologia posiada wysoko rozwinięte biblioteki, które ułatwiają pracę. Zasada przy tworzeniu serwera jest bardzo prosta. Oznaczamy klasę mapowaną na punkty końcowe poprzez adnotację *@Controller*, następnie oznaczamy poszczególne punkty końcowe poprzez adnotacje *@QueryMapping* (mapującej żądania pobierające dane) oraz *@MutationMapping* (mapującej żądania wykonujące operację).

Natomiast *graphql* wprowadza zasadniczą złożoność, która jest dodatkowym wyzwaniem dla programisty. Adnotacja *@SchemaMapping* określa mapowanie typu złożonego, który został określony w schemacie (w podanym przykładzie jest to *GroupProjection*). Mapowanie takie nazywa się *resolver* Dla wskazanego adnotacją schematu należy zaimplementować sposób pobierania danych. Zostało to przygotowane w taki sposób, aby użytkownik mógł określić, czy chce pobrać elementy, do których referencje posiada główny zasób.

Listing 14. Przykład zapytania zadeklarowanego przez użytkownika

```
query {
fetchGroupPost(groupId:”01d9ae5c-bc86-43fc-8e16-0d144e2828a4”) {
postId
author
text
attachments {
   attachmentId
   resourceLink
   type
  }
sentTime
comments
  }
}
```

Na podstawie powyższego przykładu można wytłumaczyć potencjalne problemy związane z protokołem:

- Pole *author* jest oddzielnym zasobem. Zasób ten posiada kolejne referencje. Na programiście spoczywa odpowiedzialność określenia stopnia zagnieżdżenia referencji, do których może odwołać się użytkownik
- Pole *comments* zawiera komentarze, które to także mają referencje do pobranego postu, przez co istnieje możliwość utworzenia pętli nieskończonej w wyniku cyklicznych referencji
- Zarówno sam post jak i *comment* posiadają odwołanie do profili użytkownika. Jeżeli użyto podstawowej implementacji *resolver*'a to serwer może zaciągać dane o profilach w osobnych zapytaniach (co jest określane jako problem N+1). W takim przypadku wymagane jest dostarczenie implementacji *DataLoader,* która zbiera referencje do zasobów tego samego typu, po czym pobiera dane zbiorczo.
- Pola *attachments* oraz *author* odwołują się do osobnych zasobów. Zamiast pobierać dane sekwencyjnie można zastosować asynchroniczne *resolver*'y, które choć bardziej optymalne znacznie utrudniają implementację
- Niektóre zasoby wymagają dodatkowych informacji, które mogą pojawić się w żądaniu. Informacje takie są przekazywane przez dodatkowy kontekst, którym musi zarządzać programista

Powyższe problemy optymalizacyjne wprowadzają dodatkową warstwę. Dodatkowo w celu uzyskania jak najlepszej wydajność zespół programistyczny musi nałożyć szczególny nacisk na testy wydajnościowe *API.*

Złożoność wprowadzona przez *graphql* przewyższa zalety, które oferuje w mniejszych projektach. Gdy mowa jest o aplikacji, która posiada niewielką liczbę zasobów, wybranie *graphql* prawdopodobnie okaże się niepotrzebnym nakładem pracy. Jednak, kiedy nasze *API* składa się z dziesiątek mikrousług i dziesiątek oferowanych zasobów, *graphql* potrafi ukryć złożoność interfejsu i usunąć część pracy związaną z wprowadzeniem kompozycji, oraz projektowaniem punktów końcowych dla różnych klientów.

5.2.3 Grpc

Według statystyk 20% programistów pracujących przy mikroserwisach korzysta z połączenia *RPC.* Podobnie jak w przypadku innych omawianych protokołów *grpc* posiada wsparcie w większości języków programowania. Natomiast forma samego wsparcia technologii jest inna. W przypadku *grpc* klienci do poszczególnych technologii są wygenerowani za pomocą skryptu utworzonego przez twórców *grpc.* Biblioteki udostępniające gotową implementację klientów są zatem stworzone przez samych autorów. Oprócz oficjalnych bibliotek istnieją także inne wspomagające prace z protokołem.

Korzystanie z *grpc* różni się w stosunku do innych technologii. W przypadku konkurentów programiści korzystają z bibliotek zintegrowanych ze szkieletem aplikacyjnym z którego korzystają. Natomiast *grpc* sprowadza się do wygenerowania biblioteki, która nie posiada zależności do szkieletu aplikacyjnego. Wymusza to na programiście generowanie biblioteki przy każdej zmianie schematu serwisów lub wiadomości. Wygenerowana biblioteka zawiera więcej usprawnień niż oferują to konkurenci. Biblioteka poza implementacją klientów posiada serwisy dostosowane nazwami do domeny aplikacji. Oferuje ona także projekcje zarówno dla klientów *API* jak i serwera, których zaimplementowanie jest obowiązkiem programisty w przypadku innych technologii. Dodatkowo wygenerowani klienci i projekcje wymuszają użycie odpowiednich typów danych przez osoby korzystające a *API,* co pozwala osobą implementującym serwer na pominięcie walidacji typów.

Listing 15. Przykład implementacji serwera przy wykorzystaniu *grpc*

```
class ProfileServiceGrpc(val profileService: ProfileService,
                         val imperativeSessionStorage: GRPCSessionStorage
): ProfileServiceImplBase() {
    override fun addGroupToProfile(request: AddGroupToProfileMessage?,
responseObserver: StreamObserver<Empty>?) {
        imperativeSessionStorage.userId = request!!.session.toUUID()
        profileService.addToGroup(request!!.profileId.toUUID(),
request.groupToAdd.toUUID())
        responseObserver!!.onNext(Empty.newBuilder().build())
        responseObserver.onCompleted()
    }
}
```

Na widocznym listingu klasy *ProfileServiceImplBase* i *AddGroupToProfileMessage* zostały wygenerowane przez skrypt udostępniony przez twórców *grpc*. *ProfileServiceImplBase* jest implementacją serwera, którą musimy rozszerzyć aby zaimplementować poszczególne operacje. *AddGroupToProfileMessage* jest niemutowalnym obiektem który stanowi żądanie wsyłane przez klienta. Najważniejszym elementem przykładowej metody jest *StreamObserver*. Obiekt zawiera połączenie z klientem, do którego możemy przekazać odpowiedź. W tym przypadku odpowiedź jest pusta i informuje klienta o wykonanej operacji. Odpowiedź przekazujemy poprzez wywołanie metody *onNext*. Natomiast metoda *onCompete* służy do wysłania wcześniej przekazanej odpowiedzi.

Listing 16. Przykład wykorzystania wygenerowanego klienta *grpc*

```
class ProfileServiceGRPC(val sessionStorage: SessionStorage, val
profileServiceClient: ProfileServiceBlockingStub): ProfileService {
     override fun addGroupToProfile(group: UUID, profile: UUID) {
        profileServiceClient!!.addGroupToProfile(
            com.examples.lib.AddGroupToProfileMessage.newBuilder()
                .setSession(sessionStorage.sessionOwner.userId.toString())
                .setGroupToAdd(group.toString())
                .setProfileId(profile.toString())
                .build()
        )
    }
```

Metoda korzysta z wygenerowanego klienta *ProfileServiceBlockingStub*. Użycie klienta jest bardzo proste, ponieważ nie różni się ono od użycia zwykłej metody na obiekcie. Do wywołanej metody należy przekazać żądanie w formie niemutowalnej projekcji. Zarówno użycie klienta jak i implementacja serwera *grpc* są bardzo proste i intuicyjne, ponieważ nie różnią się od użycia podstawowych funkcji języka programowania.

Generowane biblioteki z punktu widzenia dwóch zespołów tworzących oddzielne mikrousługi są bardzo przydatne. Dzięki udostępnieniu biblioteki z projekcjami programiści drugiej usługi znają szczegółowo wymagania oraz typy *API* usługi, z którą współpracują. W przypadkach innych technologii zwykle ustala się kontrakt pomiędzy zespołami tak aby te znały interfejs sąsiedniego *API*. *Grpc* znacznie ułatwia ustawienie takiego kontraktu oraz umożliwia wygenerowanie biblioteki dla języka programowania, z którego korzystają inni programiście pracujący nad sąsiednią usługą.

5.2.4 Kafka

Według statystyk JetBrains aż 47% [22] programistów pracujących z architekturą mikrousług korzysta z brokera wiadomości. Rozwiązanie często jest uważane za nieodłączną część architektury mikrousług z uwagi na możliwość wprowadzenia pełnej autonomiczności usług. Twórcy *kafki* zapewniają wsparcie dla większości języków programowania oferując gotowe biblioteki. Poza tym *kafka* oferuje także gotowe implementacje tak zwanych *connectorów*, które są gotowymi *API* umożliwiającymi zintegrowanie kafki z innymi technologiami. Poza bibliotekami oferowanymi przez twórców istnieją także te zintegrowane ze szkieletami aplikacyjnymi. Biorąc pod uwagę oferowane wsparcie oraz ilość kontrybucji do technologii kafka może być uważana za standardową technologię umożliwiającą komunikację opartą na zdarzeniach.

W omawianym projekcie użyto biblioteki *Spring for Apache Kafka*, aby zintegrować się z brokerem kafki. Biblioteka jest zbudowana na bazie adnotacji.

Listing 17. Przykład użycia biblioteki *Spring for Apache Kafka*

```
    @KafkaListener(topics = ["profile-get-request"], concurrency = "20")
    @SendTo("profile-get-response")
    fun `profile-get-request`(content: ConsumerRecord<String, String>):
Message<String>  {
        return kafkaAnswerTemplate.answer(content) {
            val map = kafkaObjectMapper.readPathVariable(content, "profileId")
            kafkaObjectMapper.convertToMessageFromBodyObject(
                profileService.fetchUserProfile(map)
            )
        }
    }
```

Z biblioteki korzystamy poprzez zaimplementowanie metody, która przyjmuję obiekt o typie *ConsumerRecord*, który zawiera odebraną od brokera wiadomość. Oraz opcjonalnie zwraca obiekt o typie *Message* który jest odpowiedzią wysłaną do wskazanego *topic'a*. Następnie na zaimplementowaną metodę nakładamy adnotację *@KafkaListener*, podajemy listę *topic'ów* które mają być nasłuchiwane przez metodę oraz opcjonalnie podajemy ilość wątków odbierających wiadomości. Jeżeli zwracamy dane w metodzie musimy także nałożyć metodę *@SendTo* podając *topic* który ma zawierać odpowiedzi.

Listing 18. Przykład wysłania wiadomości do brokera

```
override fun addGroupToProfile(group: UUID, profile: UUID) {
        objectMapper.writeValueAsString(ProfileAddedEvent(group, profile)).let {
            kafkaTemplate.send("group-profileadded-event", it)
        }
    }
```

Wysłanie wiadomości do brokera jest bardzo proste. Korzystamy z obiektu o typie *KafkaTemplate*, który jest automatycznie wygenerowany przez bibliotekę. Korzystamy z metody *send*, która przyjmuje dwa parametry. Pierwszym jest *topic* na który chcemy skierować wiadomość, a drugim jest zawartość samej wiadomości. Metoda jest przeciążona i posiada alternatywną sygnaturę, która przyjmuje obiekt o typie *ProducerRecord*. Tworząc taki obiekt do wysłania oprócz treści wiadomości oraz *topic* możemy także określić klucz wiadomości oraz nagłówki.

Oparcie komunikacji na zdarzeniach nakłada na programistów obowiązek utrzymania idempotentności oraz rozróżnianie powielonych wiadomości. Niemożliwe jest w systemie kolejkowym uzyskanie sytuacji, w której wiadomości nie będą powielane zgodnie z *Two Generals' Problem* [30]. W związku z udostępnioną konfiguracją oraz mechanizmem *ack kafka* pozwala na użycie jednej z 3 semantyk:

- Co najmniej raz (*at least once*) – semantyka zakłada, że konsument odbierze wiadomość wysłaną przez producenta przynajmniej jeden raz. Czyli nie wystąpi utracenie wiadomości. Jednak wiadomość wysłana przez producenta może być powielona lub przetworzona kilka razy. W tej semantyce konsument odbiera oraz przetwarza wiadomość, po czym informuje broker o odczytanej wiadomości. W niektórych przypadkach konsumenta powieli przetwarzanie wiadomości o ile z jakiegoś powodu nie mógł poinformować brokera o przetworzeniu wiadomości. Semantyka nie wymaga dodatkowych mechanizmów oraz nie zmniejsza wydajności systemu. Za wyborem tej semantyki decyduje to czy przetworzenie wiadomości jest idempotentne. Semantyka musi być także zachowana ze strony producenta. W tej semantyce producent ma określony czas, przez który czeka na potwierdzenie brokera o odebraniu wysłanej wiadomości Jeżeli potwierdzenie nie przyjdzie, producent wysyła wiadomość jeszcze raz.
- Co najwyżej raz (*at most once*) – konsument informuje o przetworzeniu wiadomości zaraz po jej odebraniu. Dzięki temu mamy pewność, że konsument odczytał wiadomość jedynie raz. Natomiast jeżeli wydarzy się jakiś błąd podczas przetwarzania to wiadomość zostanie nieprzetworzona i utracona. Sytuacja wygląda adekwatnie po stronie producenta. Po wysłaniu wiadomości ten nie czeka na potwierdzenie. Semantyka nie może być użyta, jeżeli dane w systemie nie mogą zostać utracone. Natomiast z uwagi na brak oczekiwania na potwierdzenie wysłania oraz niski czas oczekiwania na potwierdzenie przetworzenia wiadomości, semantyka jest najbardziej wydajną. Podejście takie dobrze się sprawdza, jeżeli odbieramy dane z receptorów, które wysyłają dane często, ale użytkownikowi zależy na najnowszych danych.
- Dokładnie raz (*exactly once*) – z definicji nie da się uzyskać takiej semantyki. Wśród specjalistów istnieje spór dotyczący *exactly once,* ponieważ część specjalistów uważa, że nie jest to semantyka, a dołożenie mechanizmów, które pozwalają na dokładnie jedno przetworzenie wiadomości. Jeżeli chodzi o komunikacje to chcąc uzyskać dokładnie jedno przetworzenie wysłanej wiadomości najpierw trzeba uzyskać semantykę *at least once.* Następnie skorzystać z jednego z proponowanych przez twórców kafki rozwiązań:
- *Transactional API* – w jednej ze swoich aktualizacji kafka wprowadziła nową bibliotekę udostępniającą transakcję. Te polegają na dodaniu do wysłanych wiadomości identyfikatora producenta, identyfikatora transakcji, identyfikatora wiadomości oraz *timestamp* wysłania wiadomości. Informacje są uzupełniane automatycznie natomiast na programiście spoczywa obowiązek określenia poziomu izolacji transakcji oraz otwieranie i zamykanie transakcji. Korzystanie z *Transactional API* jest podobne do korzystania z transakcji relacyjnej bazy danych. Jednak w przypadku *kafki rollback* jest określany przez programistę ręcznie. Sama biblioteka zaproponowana przez *kafkę* nie jest jednak wystarczająca. Transakcje zakładają, że dane są odczytywane z jednego *topic'a* i zapisywane na drugi. W takim przypadku faktycznie da się zapewnić przetwarzanie dokładnie jeden raz. Natomiast jeżeli w środku transakcji *kafki* dane zapisujemy do innego systemu lub bazy danych, to trzeba wyposażyć system w alternatywne rozwiązanie.
- Rejestrowanie identyfikatorów wiadomości – podczas wiadomości można dopisać do nagłówka *kafki* unikatowy identyfikator wiadomości. Konsument korzysta z dodatkowej bazy danych w której zapisuje przetworzone już identyfikatory wiadomości. Po rozpoczęciu przetwarzania otwieramy transakcję w bazie danych. Jeżeli wydarzy się awaria transakcji zapisu identyfikatora ostanie odwrócona. Kafka także nie zostanie poinformowana o przetworzeniu wiadomości, dzięki czemu nie tracimy danych. Jeżeli jednak transakcja powiedzie się i otrzymamy duplikat wiadomości to zostanie on odrzucony, ponieważ przed każdym przetworzeniem wiadomości sprawdzamy czy jej identyfikator widnieje już w lokalnej bazie danych.

Każde z rozwiązań nakłada na programistę dodatkowe obowiązki oraz znacznie zmniejsza wydajność z powodu zarządzania transakcjami.

Z perspektywy programisty korzystanie z *kafki* jest bardzo skomplikowane i stawia wyzwania nieobecne w pozostałych omawianych technologiach. Programista musi sprecyzować, czy operacja jest idempotentna, określić semantykę jaka jest korzystna i zaimplementować mechanizm transakcji, jeżeli istnieje taka potrzeba. Dodatkowo programista musi także określić ilość wątków przypadających na jednego konsumenta. W żadnej innej z omawianych technologii programista nie musiał podejmować tylu decyzji znacznie wpływających na wydajność systemu.

5.3. Wpływ technologii na komunikacje klient-serwer

Niektóre z opisanych technologii wpływają na sposób w jaki klient komunikuje się z systemem. Klient chcący zintegrować się z systemem musi dostosować się do udostępnionego *API*. Protokół jaki narzuca system jest oceniony pod względem wprowadzonej złożoności oraz użyteczności z punktu widzenia aplikacji klienckiej. Wzięto pod uwagę również praktyki, które są używane w czasie współpracy z daną technologią, aby wyróżnić dodatkowe obowiązki, jeśli API ma być użyteczne dla odbiorcy.

5.3.1 HTTP

Protokół *http* jest podstawową technologią używaną do komunikacji między aplikacją kliencką a serwerem. Był on doskonalony przez lata, przez co ciężko o lepszą alternatywę. Protokół jest bardzo prosty, klient chcący zmodyfikować lub pobrać zasób opisuje go w formie czasownika, nagłówków i ciała, po czym wysyła żądanie i czeka na odpowiedź. Dodatkowo protokół został wyposażony w komunikację za pomocą strumienia danych oraz liczne usprawnienia.

Chociaż protokół jest najbardziej rozpowszechniony to okazuję się, że posiada kilka znaczących ograniczeń:

- Brak możliwości strumieniowania danych wysyłanych przez klienta do serwera
- Brak strumieniowania dwukierunkowego w którym dane mogą być wysłane zarówno przez klienta jak i serwer
- Elastyczny format *JSON* służący do przesyłania informacji nakłada na zespół pracujący z protokołem udostępnienie dokumentacji oraz schematu poszczególnych zapytań. (Rozwiązaniem tego jest technologia *OpenApi* omówione poniżej)
- Brak możliwości szczegółowego określenia potrzebnych danych, co często wymusza wysłanie kilku zapytań do tego samego serwera w celu uzyskaniu kilku zasobów

Powyższe ograniczenia doprowadziły do powstania kilku alternatyw zbudowanych na podstawie protokołu *http*. Rozwiązania takie jak *grpc* oraz *graphql* zostały szczegółowo omówione w innych rozdziałach

Protokół *http* wymaga jednak dodatkowych elementów, bez których klienci nie są w stanie korzystać z zasobów wystawionego przez nas API. Protokół zakłada, że udostępnimy klientowi szereg linków z zasobami. Każdy zasób jest opisany przez adres *URL*, czasownik *http* oraz parametry zapytania. Daje to dużą dowolność implementacji, co doprowadziło do kilku problemów.

Każdy twórca API może wyrobić swoje własne podejście do określania poszczególnych operacji przez parametry zapytania. Klient chcący korzystać z API musi najpierw nauczyć się podejścia do nazewnictwa oraz schematu zapytań stworzonego przez twórcę. Takie podejście jest toporne z punktu widzenia użytkownika API, natomiast różnice w koncepcjach nazewnictwa *API* w obrębie jednego systemu są niespójnościami, które utrudniają proces wytwarzania oprogramowania. Całość tych problemów rozwiązano poprzez wprowadzenie architektury *REST* oraz 4 poziomu

dojrzałości *REST API*. 4 poziomy dojrzałości opisują koncepcje, które powinny zostać użyte podczas projektowania *API* przez jego twórcę:

- Poziom 0 – określa sposób nazewnictwa linków, które wskazują na zasoby. Sprowadza się to do zakazania znaków specjalnych w linkach
- Poziom 1 - każdy link rozpoczyna się od głównego zasobu, a kolejne człony linku podzielone przez znak „/" określają *podzasób* zasobu oraz identyfikator pojedynczego obiektu. Przykładowo link „*group/2/profile"* wskazuje na listę profili przypisanych do grupy o identyfikatorze 2
- Poziom 2 – określa w jaki sposób stosować resztę atrybutów żądania *http,* czyli nagłówki, parametry zapytania, czasowniki oraz statusy odpowiedzi
- Poziom 3 – zawiera specyfikacje bardziej abstrakcyjnych elementów takich jak wersjonowanie, negocjacja formatu (*content negotiations*) oraz *HATEOAS*. Wersjonowanie często jest wykorzystywane przez programistów, jednak *HATEOAS* i *content negotiations* zwykle jest pomijane z uwagi na małą użyteczność.

W wyniku dużej swobody danej twórcom *API* zobowiązano ich do stosowania w swoich systemach powszechnych standardów. Oprócz zapewnienia spójności programiści muszą także udokumentować *API*. Dokumentacja zwykle tworzona jest przez zintegrowanie systemu z technologią *OpenAPI*. [31] Dodanie tej technologii do projektu wymaga wprowadzenia kolejnych zmian na utworzone już punkty końcowe.

Niestety swoboda protokołu *http* okazała się jego wadą, która w rezultacie nałożyła kolejne obowiązki na programistów, którzy muszą dostosować kod do określonych koncepcji oraz wprowadzać zmiany związane z dokumentacją. Integrowanie aplikacji z dodatkową technologią wprowadza dodatkowe koszty na kilku etapach wytwarzania oprogramowania. Kiedy programista skończy implementację musi nanieść dodatkowe elementy wymagane przez koncepcję i dokumentację. Całość opisu punktów końcowych *API* jest następnie weryfikowana przez testerów oraz sprawdzana przez analityków. Generowanie dokumentacji najczęściej musi być zautomatyzowane co wprowadza kolejny krok do procesów *Continous Integration* i *Continous Delivery*. Dodatkowy nakład pracy związany ze specyfikacją i utrzymaniem tak zwanego kontraktu powinien być brany pod uwagę przy podejmowaniu decyzji o używanym protokole.

Jeśli chodzi o autoryzację ta może być realizowana na kilka sposobów. Zwykle sprowadza się to do zamieszczenia wygenerowanego przez system autoryzujący tokenu. Token jest umieszczony w nagłówkach zapytania. Kolejnym elementem związanym z bezpieczeństwem jest utajnienie danych przesyłanych pomiędzy klientem, a aplikacją. Przesyłane dane są szyfrowane kluczem uzgodnionym przez dwie strony w wyniku *SSL/TLS Handshake,* a sam protokół jest określany wtedy jako *https (Hypertext Transfer Protocol Secure).* Dodatkowa warstwa związana z szyfrowaniem jest w pełni zautomatyzowana z perspektywy klienta i serwera. Dodatkowym obowiązek jest skonfigurowanie połączenia poprzez skonfigurowanie klucza publicznego po stronie serwera oraz uzyskanie certyfikatu autoryzującego wydanego przez specjalną instytucję.

5.3.2 Graphql

Graphql został zaprojektowany, aby usprawnić komunikacje pomiędzy klientem, a serwerem. Poprzednik *graphql* czyli *http* miał kilka ograniczeń, z których większość rozwiązuje *graphql*:

- *Http* nie posiadał integralnego rozwiązanie, jeśli chodzi o typ danych przekazywanych serwerowi. *Graphql* pozwala dokładnie określić typy przesyłanych informacji
- *Graphql* w przeciwieństwie do *http* pozwala na określenie danych, które użytkownik chce otrzymać od aplikacji. Z punktu widzenia klienta jest to intuicyjne, użytkownik określa zasoby, referencje i poszczególne pola, które chce otrzymać. Dla przykładu w *http* użytkownik musiałby poznać kilka punktów końcowych po czym odwoływać się do nich w kolejności zgodnej z otrzymanymi danymi. Rezultatem podejścia *http* jest narzucenie na użytkownika optymalizacji

wysyłanych zapytań. Definiowanie grafu przez serwer pozwala na usprawnienie komunikacji klient-serwer z dużym zyskiem, jeśli chodzi o latencje końcową (co zostało udowodnione w rozdziale 5.4)

Prócz usprawnienia komunikacji klient-serwer *graphql* posiada następujące zalety:

- Autoryzacja odbywa się w taki sam sposób jak w przypadku *http*
- Szyfrowanie odbywa się przy pomocy *TLS/SSL* podobnie jak w przypadku *http*
- Protokół udostępnia funkcje, które oferował http np. strumień danych przesłanych od serwera do klienta

Mimo dużych usprawnień oferowany dla użytkowników *API, graphql* także posiada ograniczenia, które posiada *http*. Protokół nie daje możliwości dwukierunkowego strumieniowania danych oraz nie umożliwia strumieniowania danych od klienta do serwera.

5.3.3 Kafka

Kafka może zostać zintegrowana z protokołem *http* na dwa omówione wcześniej sposoby. Rozwiązanie z użyciem dwóch kolejek i blokowaniu użytkownika nie zmienia sposobu w jaki klient integruje się z aplikacją. Użytkowanie systemu staje się zbieżne z tym co oferuje protokół *http* lub *graphql*. Istnieje jednak możliwość wydzielenia takich operacji (*jak np. dodanie do grupy*), które nie muszą być w całości wykonywane podczas komunikacji z klientem. Dzięki *kafce* można oddelegować przetwarzanie operacji i poinformować klienta jedynie o przyjęciu wiadomości.

Drugim sposobem jest reaktywne powiadomienie klienta o wykonaniu operacji i przekazanie mu wyniku. Można to zrobić przy użyciu *Server Sent Events*. Rozwiązanie jednak jest bardzo inwazyjne i wymusza na kliencie implementacje mechanizmu oczekiwania na odpowiedź.

Listing 19. Przykład klienta łączącego się z aplikacją przy użyciu *SSE*

```
#Wysłanie wiadomości
headers = PARAMS["HEADERS"].copy()
r = requests.get(url=PARAMS["URL"], headers=headers, json=PARAMS["JSON"])

#oczekiwanie na odpowiedź nasłuchując strumień danych
http = urllib3.PoolManager()
headers_copy = PARAMS["HEADERS2"].copy()
stream_response = http.request('GET', "http://localhost:9090/sse",
preload_content=False, headers=headers_copy)
client = sseclient.SSEClient(stream_response)
for event in client.events():
    print(event.data)
client.close()
```

Implementacja jest podzielona na dwie części. Pierwszą jest wysłanie operacji, a drugą stworzenie strumienia danych pomiędzy serwerem a klientem. Klient nasłuchuje następnie na zdarzenie, które są strumieniowane przez serwer. Jednym ze zdarzeń powinien być efektem wcześniejszego zapytania. Rozwiązanie takie nakłada dodatkową pracę związaną z otworzeniem strumienia danych oraz odbieraniem wiadomości (zwykle odbywającym się w osobnym wątku). Z punktu widzenia klienta rozwiązanie takie nie daje korzyści, a obciąża aplikacje kliencką.

Kolejnym problemem, jest *eventual consistency* które powstaje podczas operacji. Użytkownik zwykle oczekuje, że wprowadzenie operacje da natychmiastowy efekt po stronie interfejsu. Jeżeli dane mutowane podczas operacji są widoczne na ekranie użytkownika to należy zaprojektować operacje tak aby ich cześć była wykonywana tuż po otrzymaniu wiadomości. Jeżeli operacja nie jest podzielna

można wykonać ją natychmiastowo rezygnując z przetwarzania reaktywnego lub zablokować klienta na odpowiednio długi czas (pokazując mu na interfejsie graficznym grafikę ładowania). Najcięższa sytuacja, w której operacja jest długa niestety równa się z poszukiwaniem kompromisu i zaprojektowaniu aplikacji klienckiej tak aby podział na operacje był widoczny dla użytkownika.

Inwazyjność *kafki* zależy głównie od projektu systemu. Jeżeli nasz system będzie opierał wszystkie operacje na kolejkowaniu, a *API* będzie wymagać strumieniowania to nasz system będzie bardzo problematyczny w obsłudze. Jednak dobre wykorzystanie kolejkowania może dać duże zalety z punktu widzenia klienta. Oddelegowania długich operacji zmniejsza czas zablokowania klienta. Możliwe jest także nasłuchiwanie zdarzeń tak aby na bieżąco widzieć status długich operacji. System będzie przyjmował także zlecenia, mimo że część systemu odpowiedzialna za nie będzie niedostępna. Użytkownicy będą mieli także wrażenie, że korzystają z systemu o dużo większej przepustowości, ponieważ nadmiar żądań będzie kolejkowany. Zalety te wchodzą w skład bardzo ważnej cechy systemu jaką jest niezawodność.

5.4. Analiza wydajnościowa

Analiza jest efektem badań przeprowadzanych na poszczególnych wariantach aplikacji społecznościowej. Prezentowane wyniki są efektem testów wydajnościowych przeprowadzonych na tych samych zasobach możliwie w jak najbardziej podobny sposób. Testowanie niektórych technologii będzie zawierało dodatkowe przypadki w celu zbadania ich cech szczególnych. Niektóre badania zostały przeprowadzone kilka razy po wprowadzeniu potrzebnych optymalizacji, co także zostało opisane. Głównymi czynnikami branymi pod uwagę jest końcowa latencja (czyli czas oczekiwania klienta na odpowiedź) oraz czas zablokowania klienta (czas poświęcony na wysłanie wiadomości). Poza wynikami została także opisana reakcja poszczególnych technologii na zbyt duże obciążenie systemu.

Testy zostały przeprowadzone na kilku operacjach oferowanych przez przykładowy system. Operacje wybrano w taki sposób, aby odzwierciedlały różne przypadki komunikacji:

- Dodanie użytkownika do grupy – jest to wywołanie operacji na obiekcie znajdującym się w systemie. Głównym celem z punktu widzenia wydajności jest jak najkrótsze zablokowanie użytkownika oraz poinformowanie go o efekcie
- Zdarzenie emitowane przez *PostApp* do *ProfileApp* informujące o zmianie sieci użytkownika – informacja ta jest czysto techniczna. W tym przypadku ważniejsze jest jak najkrótsze zablokowanie usługi wysyłającej zdarzenie niż latencja końcowa przesłanej wiadomości
- Utworzenie posta – operacja, która tworzy nowy post przekazany przez użytkownika. Odbywa się to z udziałem tylko jednej usługi
- Pobranie *Feed* – użytkownik chcący pobrać listę nowych postów, które pojawiły się w jego sieci wywołuje szereg operacji wymagających komunikacji pomiędzy usługami
- Pobranie informacji o sieci społecznościowej przez *FeedApp* znajdującej się w *ProfileApp* - aby poprawnie wygenerować *feed* usługa musi znać sieć społecznościową, która jest tworzona przez *ProfileApp*
- Pobranie postów – standardowe zapytanie pobierające dane z serwera. W tym przypadku jest to zbiór postów, które udostępnił wskazany użytkownik

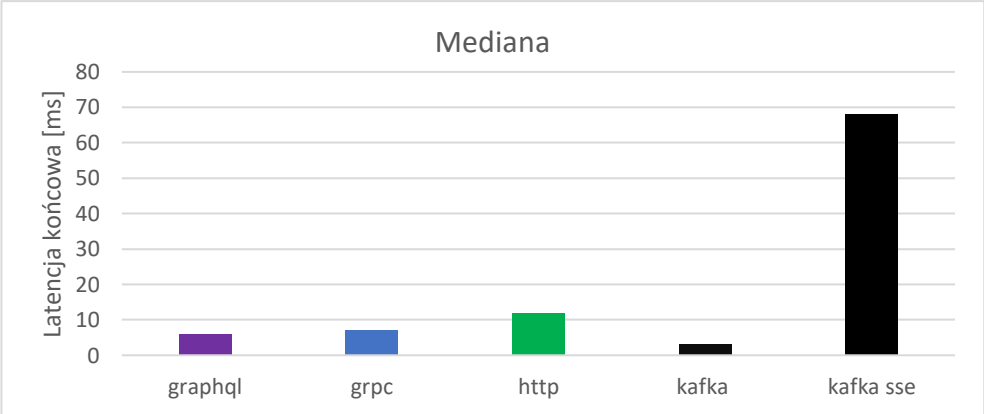
We wszystkich powyższych przypadkach testy przeprowadzano dwukrotnie. W pierwszym podejściu wysyłano 100 żądań z 10 sekundowym odstępem, tak aby sprawdzić minimalną możliwą do uzyskania latencję. Przy drugim podejściu wysyłano jednocześnie 10000 wiadomości, tak aby sprawdzić zachowanie danej technologii przy większym obciążeniu. Widniejące latencje będą medianą uzyskaną z zebranych wyników.

W przypadku *graphql* testy zostały ograniczone do komunikacji zewnętrznej (pomiędzy klientem a serwerem) z przyczyn opisanych w podrozdziale dotyczącym *graphql*. Natomiast dodano dodatkowe scenariusze porównujące technologie *graphql* zarówno z *http* jak i *grpc:*

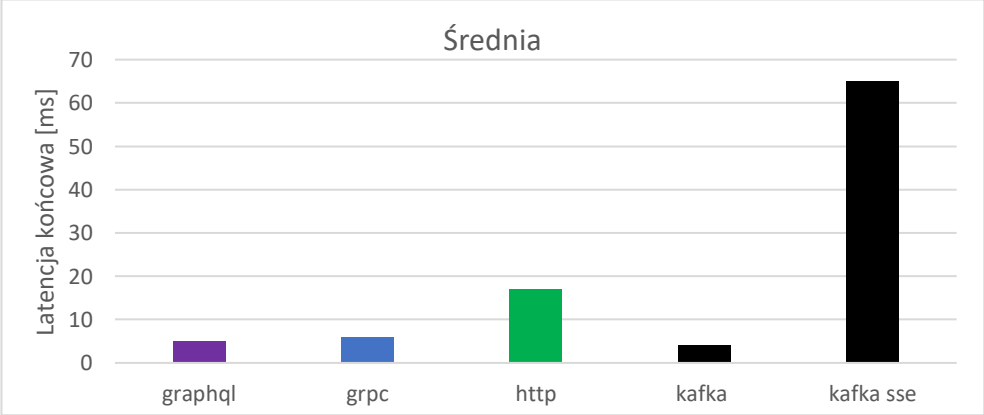
- Pobranie ograniczonych danych o grupie – w przypadku *graphql* użytkownik decyduje o wartościach, które chce pobrać. Należy sprawdzić czy takie ograniczenie wpłynie na latencję
- Pobranie danych wskazanych przez referencję – w przypadku *graphql* użytkownik może pobrać za jednym żądaniem kilka typów zasobów, co nie jest możliwe w przypadku pozostałych technologii

5.4.1 Dodanie do grupy

W pierwszym kroku zbadano latencję końcową dla operacji dodania nowego użytkownika przy małym obciążeniu systemu (rysunek 9 i 10).



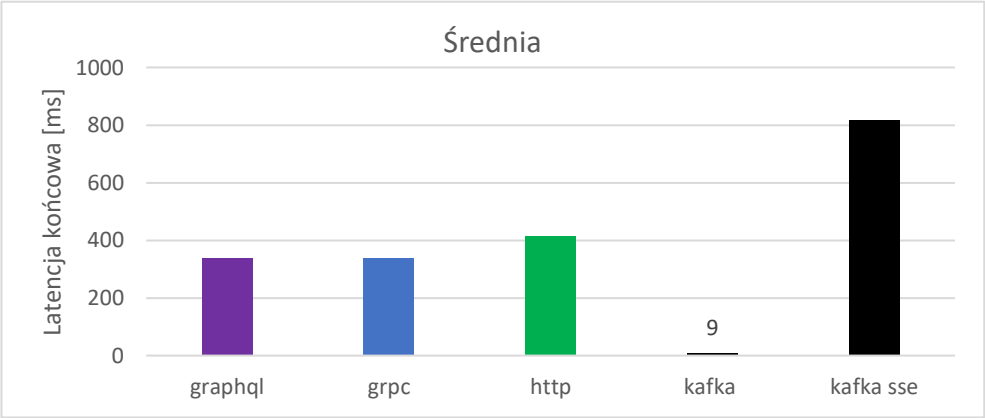
Rysunek 9. Wykres mediany latencji końcowej dla operacji dodania użytkownika przy małym obciążeniu (lepsza jest wartość mniejsza)



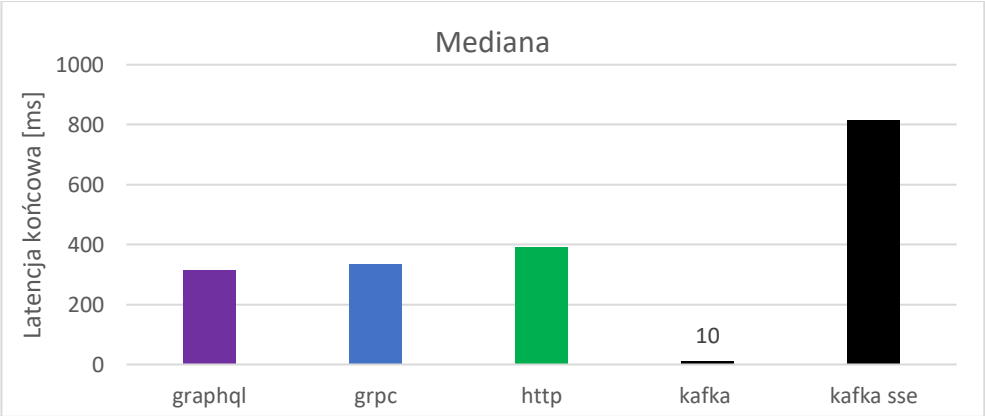
Rysunek 10. Wykres średniej latencji końcowej dla operacji dodania użytkownika przy małym obciążeniu (lepsza jest wartość mniejsza)

Wyniki dla *grpc* oraz *graphql* są bardzo zbliżone. Z powodu *load balancingu* pierwsze zapytania *http* mają wyższą latencje jednak ta jest niska dla większości użytkowników (rysunek 10). Wyróżnia się natomiast bardzo duża latencja *kafka sse* która jest czasem pomiędzy początkiem wysłania żądania a odebraniem odpowiedzi przesłanej przy pomocy strumienia danych (*Server Sent Events*) [32]. Powstała latencja wynika z faktu wielokrotnego konwertowania danych. Dane przesłane

są najpierw przekształcane z protokołu *http* na obiekt systemu a następnie ponownie konwertowane na wiadomość wysłaną do *kafki*. Wiadomość w po odebraniu jest ponownie konwertowana na obiekt i po przetworzeniu wysyłana jako wiadomość zawierająca odpowiedź systemu. Odpowiedź jest odbierana przez usługę, która utrzymuje strumień danych z klientem. Identyfikator sesji jest który znajduje się w wiadomości jest odczytywany i służy do odnalezienia właściwego strumienia danych. Oczywiście dane przed wysłaniem przez strumień ponownie muszą być przekonwertowane. Tak duża ilość dodatkowych operacji nieobecna w pozostałych technologiach skutkuje bardzo dużą latencją końcową. Rysunek 11 oraz rysunek 12 przedstawiają wyniki dla testu przy dużym obciążeniu.



Rysunek 11. Wykres średniej latencji końcowej dla operacji dodania użytkownika przy dużym obciążeniu (lepsza jest wartość mniejsza)

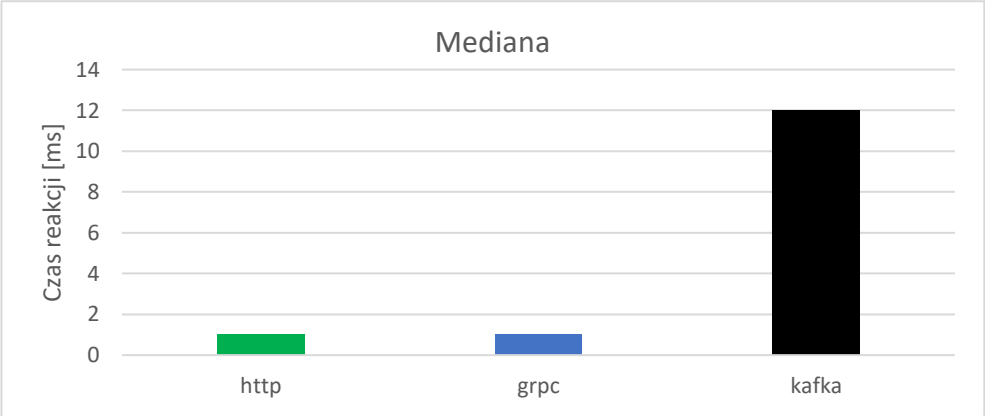


Rysunek 12. Wykres mediany latencji końcowej dla operacji dodania użytkownika przy dużym obciążeniu (lepsza jest wartość mniejsza)

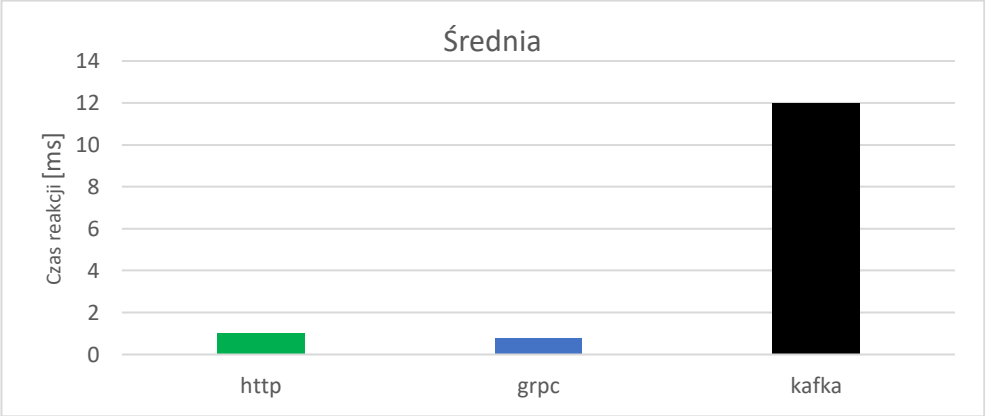
Wyniki są wyraźnie wyższe niż w przypadku niskiego obciążenia. Jednak latencja dla *grpc*, *graphql* oraz *http* zawiera niewielką różnicę, która jest zbyt mała, aby wyciągnąć na jej podstawie wnioski. Natomiast *kafka*, która pozwala nie blokować klienta pozostała na tym samym poziomie latencji (10 ms), co w przypadku niskiego obciążenia. Podobnie *kafka sse* ma znacznie większą latencję wynikającą z dodatkowych operacji podjętych w celu poinformowania użytkownika o efekcie zapytania. Dodatkowe mechanizmy muszą być skalowane wspólnie z resztą systemu w celu obsłużenia większego ruchu sieciowego.

5.4.2 Zdarzenie dodania do grupy

Kolejną operacją braną pod uwagę jest zdarzenie emitowane przez usługę *Post* i odbierane przez usługę *Profile*. Wyniki zostały podzielone na dwie grupy z uwagi na charakter połączenia. Pierwsza grupa zawiera czas reakcji, czyli czas pomiędzy wysłaniem wiadomości, a odebraniem jej przez usługę docelową. Natomiast druga grupa zawiera czas zablokowania klienta emitującego zdarzenie.

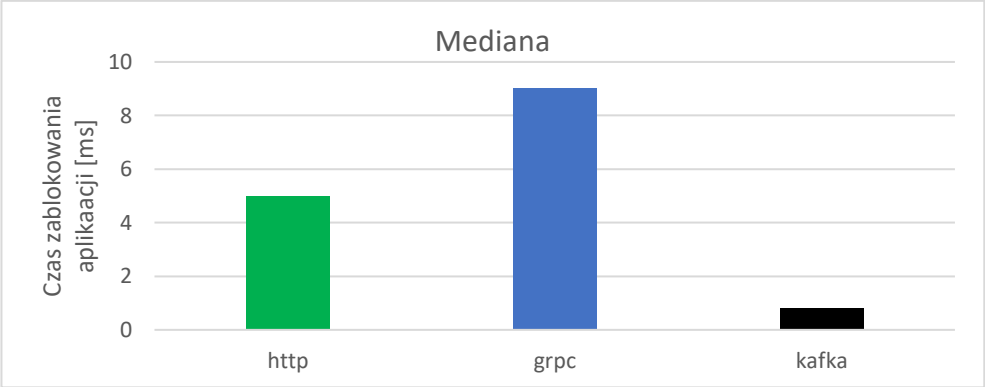


Rysunek 13. Wykres mediany czasu reakcji na wyemitowane zdarzenie przy małym obciążeniu (lepsza jest wartość mniejsza)

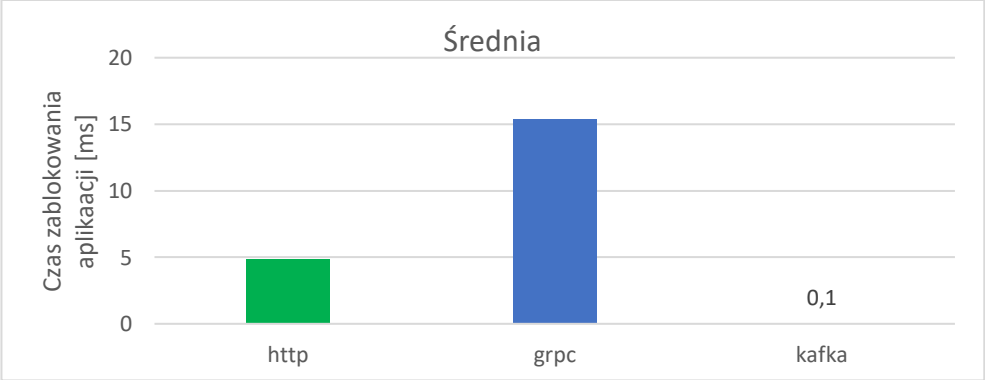


Rysunek 14. Wykres średniej czasu reakcji na wyemitowane zdarzenie przy małym obciążeniu (lepsza jest wartość mniejsza)

Czas reakcji dla *kafki* jest wyższy z uwagi na system *batch'y*, który czeka na określoną liczbę rekordów w celu przetworzenia ich zbiorczo. Dodatkowo *kafka* przed poinformowaniem konsumenta o nowej wiadomości zapisuje ją do pamięci lokalnej, co także mogło zwiększyć latencję. Jednak wyniki czasu zablokowania klienta przedstawione na rysunku 15 i 16 prezentują inny trend.

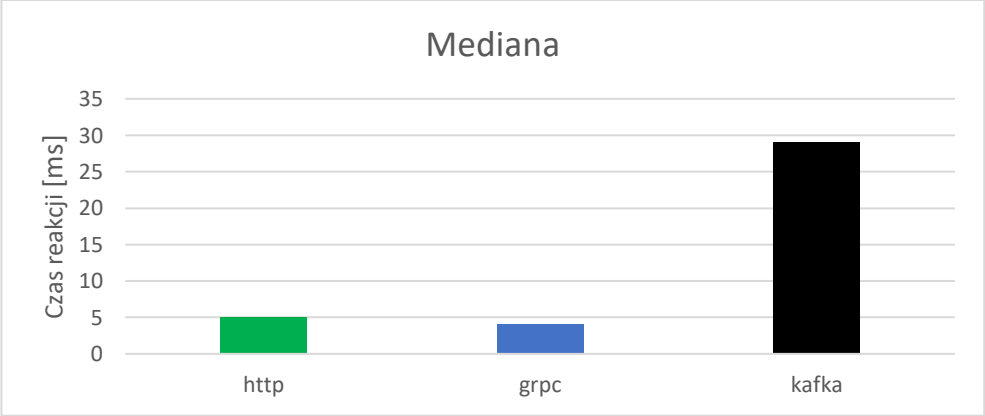


Rysunek 15. Wykres mediany czasu zablokowania klienta emitującego zdarzenie przy małym obciążeniu (lepsza jest wartość mniejsza)

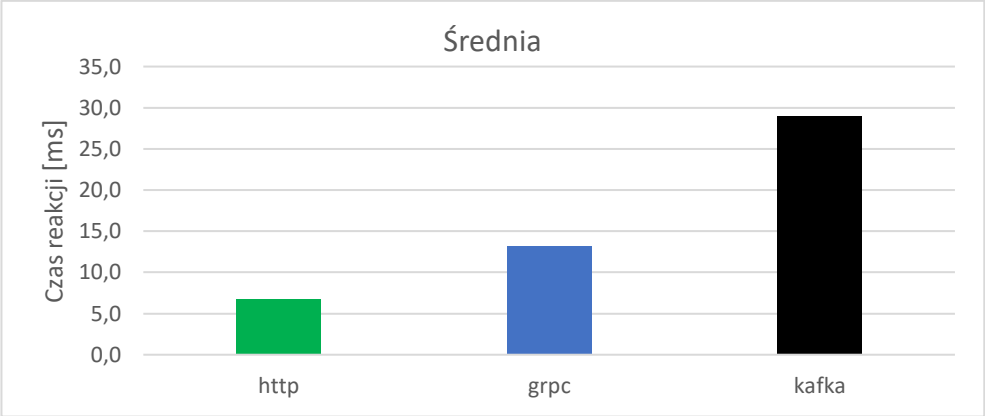


Rysunek 16. Wykres średniej czasu zablokowania klienta emitującego zdarzenie przy małym obciążeniu (lepsza jest wartość mniejsza)

Najwyższy czas zablokowania klienta posiada *grpc*, choć jest on jedynie większy o kilka milisekund. *Kafka* z uwagi na nieblokujący charakter komunikacji oraz korzystanie z niskopoziomowego protokołu *TCP* była w stanie uzyskać czas 0.8 milisekundy. Rysunek 17 i 18 przedstawiają rezultat uzyskany przy dużym obciążeniu.

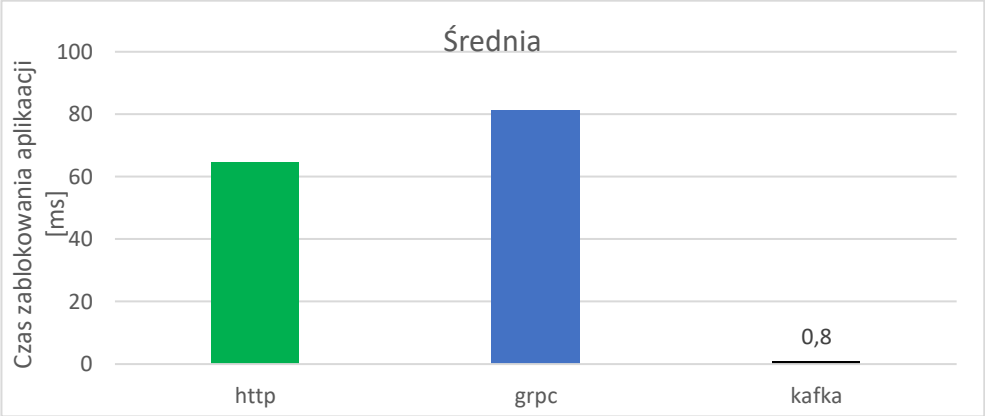


Rysunek 17. Wykres mediany czasu reakcji na wyemitowane zdarzenie przy dużym obciążeniu (lepsza jest wartość mniejsza)

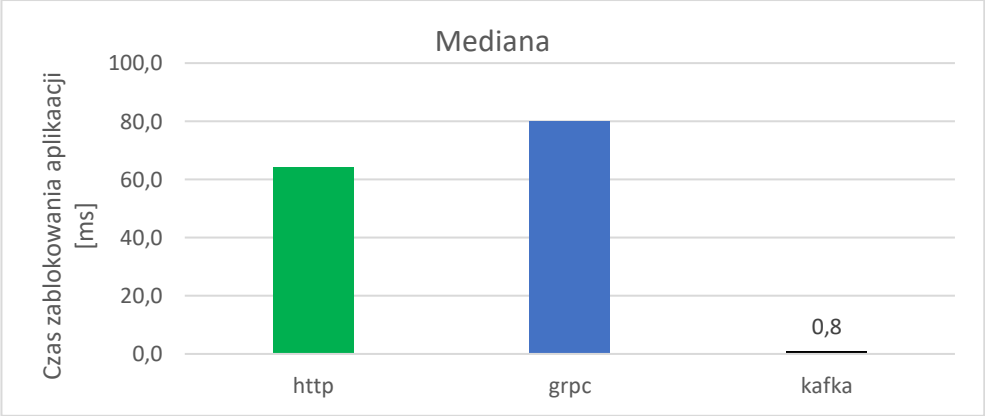


Rysunek 18. Wykres średniej czasu reakcji na wyemitowane zdarzenie przy dużym obciążeniu (lepsza jest wartość mniejsza)

Czas reakcji przy większym obciążeniu wydłużył się w przypadku każdej z technologii. Natomiast różnice pomiędzy poszczególnymi technologiami są podobne.



Rysunek 19. Wykres średniej czasu zablokowania klienta emitującego zdarzenie przy dużym obciążeniu (lepsza jest wartość mniejsza)

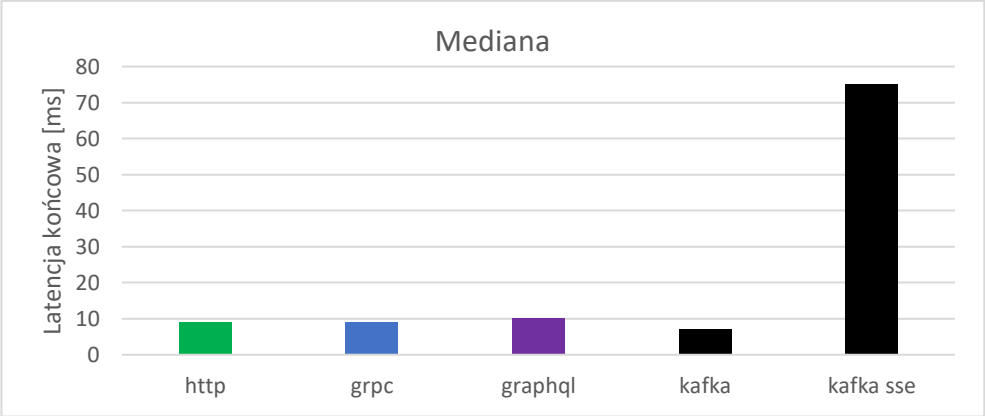


Rysunek 20. Wykres mediany czasu zablokowania klienta emitującego zdarzenie przy dużym obciążeniu (lepsza jest wartość mniejsza)

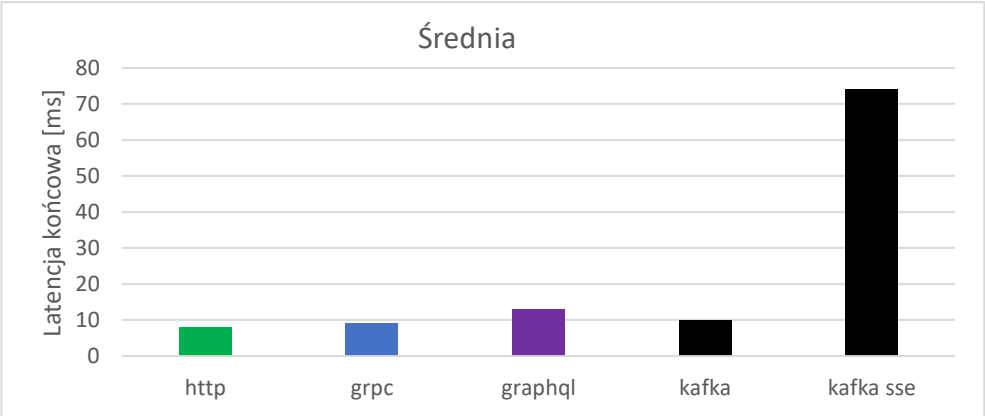
Czas zablokowania klienta zmienił się w przypadku *http* oraz *grpc*, ponieważ technologie te korzystają z blokującego stylu komunikacji. Czas nie zmienił się w przypadku *kafki* i wyniósł 0.8 milisekundy. Biorąc pod uwagę cel przysyłania zdarzenia najlepiej wypadła *kafka*. Mimo, że czas reakcji jest znacznie większy, to system nie jest obciążony zablokowanymi wątkami, co w szerszej perspektywie przekłada się na brak potrzeby skalowania serwisu emitującego zdarzenie. Większy czas reakcji *kafki* również przy dużym obciążeniu jest spowodowany zbyt małą ilością wątków i partycji obsługujących zdarzenia. Przy powiększeniu tych liczb byłoby możliwe zmniejszenie czasu reakcji z 30 milisekund na 12 milisekund. Chcąc zmniejszyć czas zablokowania klienta w przypadku *http* oraz *grpc* trzeba wprowadzić mechanizm oddelegowania pracy do puli wątków, co sprowadza się do odtwarzania funkcjonalności oferowanych przez *kafkę*. Drugim sposobem na zmniejszenie czasu zablokowania klienta jest proporcjonalne skalowanie usługi *Profile* do usługi *Post*. W przypadku *kafki* usługi nie muszą być skalowane proporcjonalnie z uwagi na brak sprzężenia.

5.4.3 Stworzenie postu

Kolejnym porównywaną operacją jest utworzenie postu. Zmierzono latencje przy małym obciążeniu systemu. Wyniki z kolejnych testów widoczne są na rysunkach 21 i 22.

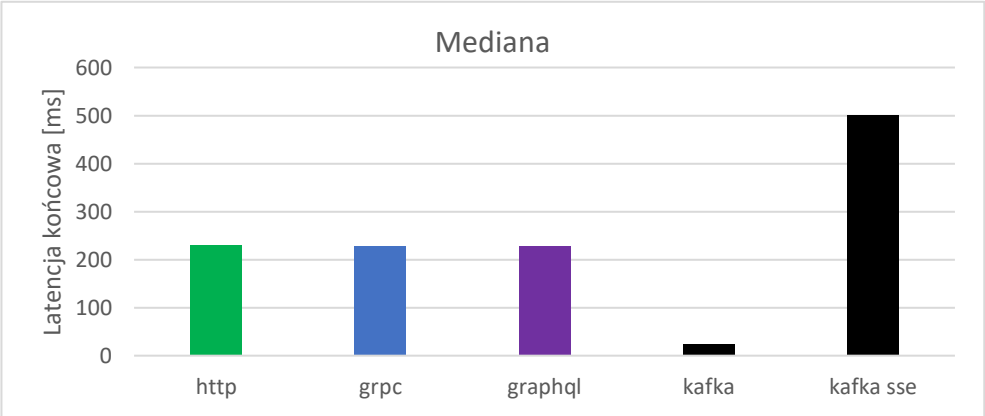


Rysunek 21. Wykres mediany latencji końcowej dla operacji stworzenia posta przy małym obciążeniu (lepsza jest wartość mniejsza)

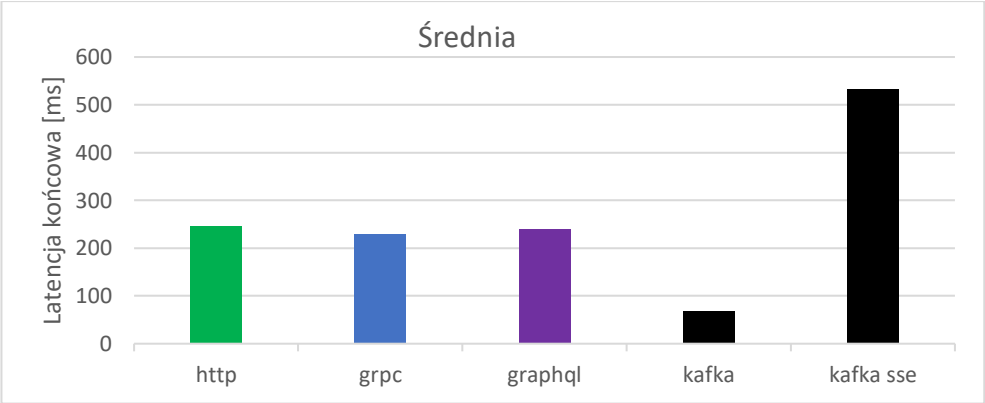


Rysunek 22. Wykres średniej latencji końcowej dla operacji stworzenia posta przy małym obciążeniu (lepsza jest wartość mniejsza)

Wyniki są podobne do operacji dodania użytkownika do grupy. Technologie *graphql*, *http* oraz *grpc* mają bardzo zbliżone rezultaty, a różnica pomiędzy nimi jest pomijalna. Natomiast czas odpowiedzi *kafka sse* jest ponownie bardzo duży. Trend jest zachowany również przy większym obciążeniu, co jest widoczne na rysunkach 23 i 24.



Rysunek 23. Wykres mediany latencji końcowej dla operacji stworzenia posta przy dużym obciążeniu (lepsza jest wartość mniejsza)

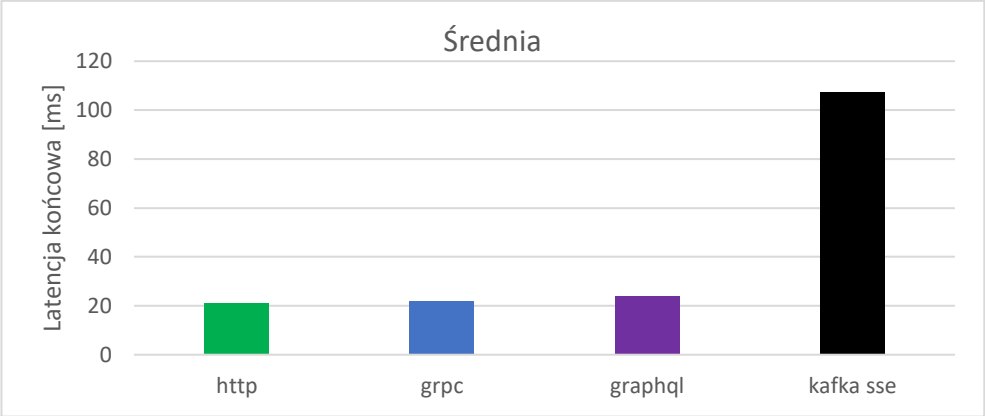


Rysunek 24. Wykres średniej latencji końcowej dla operacji stworzenia posta przy dużym obciążeniu (lepsza jest wartość mniejsza)

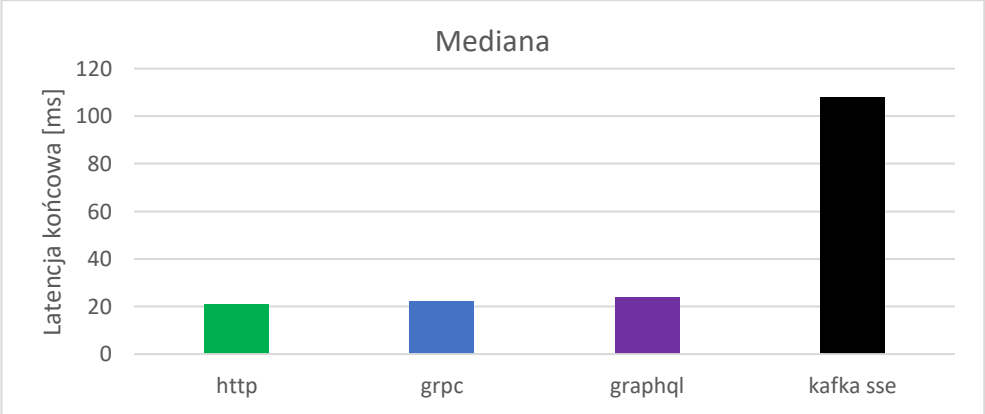
Przy większym obciążeniu wyniki są wyższe natomiast nie uwydatniły się różnice pomiędzy technologiami.

5.4.4 Pobranie *feed*

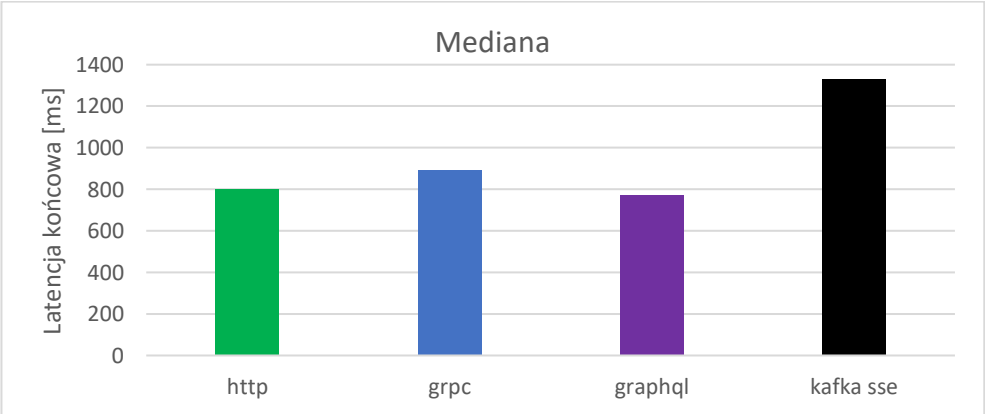
Kolejną mierzoną operacją jest pobranie nowych postów użytkownika, czyli *feed*. Wyniki zarówno dla dużego i małego obciążenia wyglądają podobnie do operacji pobrania postu i dodania użytkownika do grupy. Większa latencja końcowa w przypadku tej operacji wynika z jej złożoności.



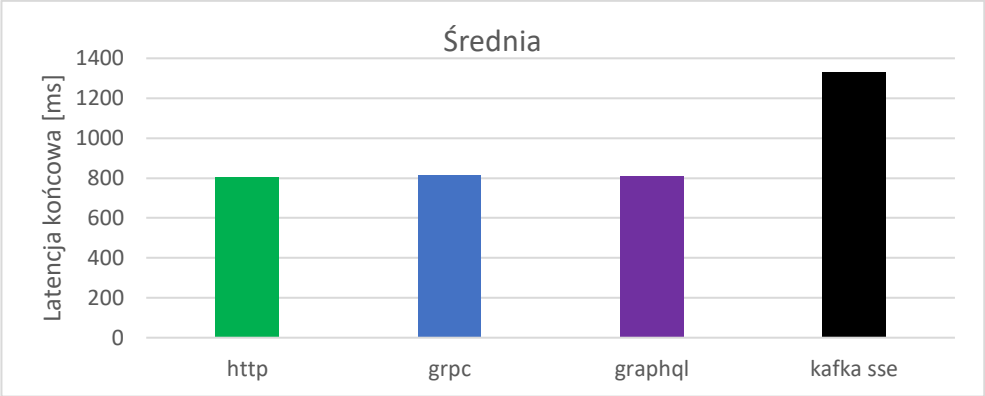
Rysunek 25. Wykres średniej latencji końcowej dla operacji pobrania *feed* przy małym obciążeniu (lepsza jest wartość mniejsza)



Rysunek 26. Wykres mediany latencji końcowej dla operacji pobrania *feed* przy małym obciążeniu (lepsza jest wartość mniejsza)



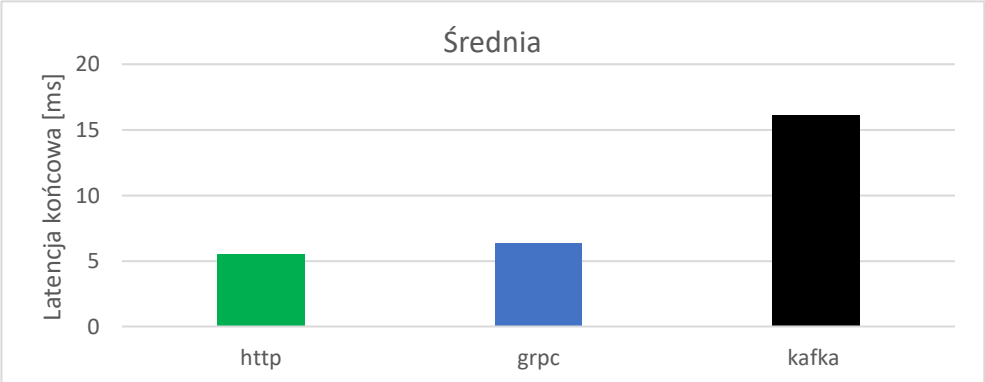
Rysunek 27. Wykres mediany latencji końcowej dla operacji pobrania *feed* przy dużym obciążeniu (lepsza jest wartość mniejsza)



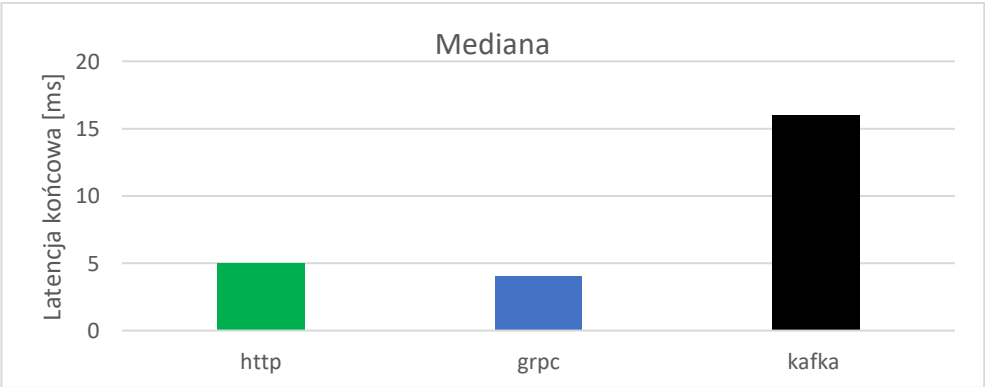
Rysunek 28. Wykres średniej latencji końcowej dla operacji pobrania *feed* przy dużym obciążeniu (lepsza jest wartość mniejsza)

5.4.5 Pobranie sieci społecznościowej

Podczas przetwarzania operacji pobrania *feed* mikroserwis *Feed* musi pobrać informacje o sieci użytkownika od serwisu *profile*.

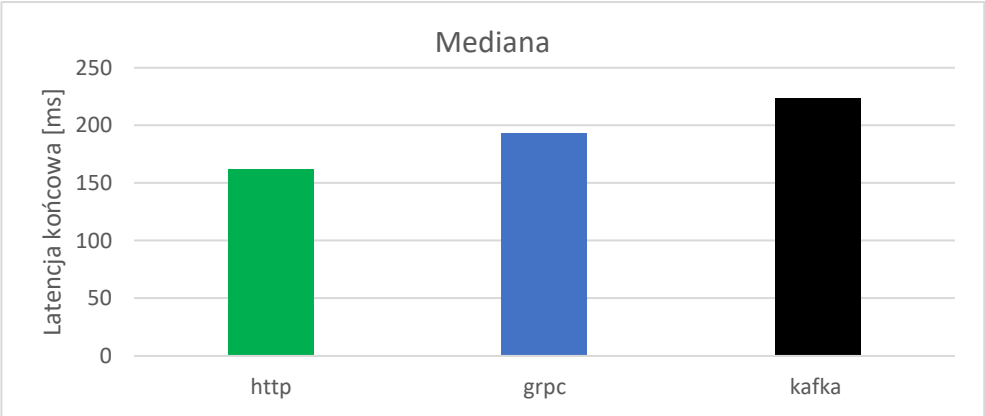


Rysunek 29. Wykres średniej latencji końcowej dla pobierania sieci społecznościowej przy małym obciążeniu (lepsza jest wartość mniejsza)

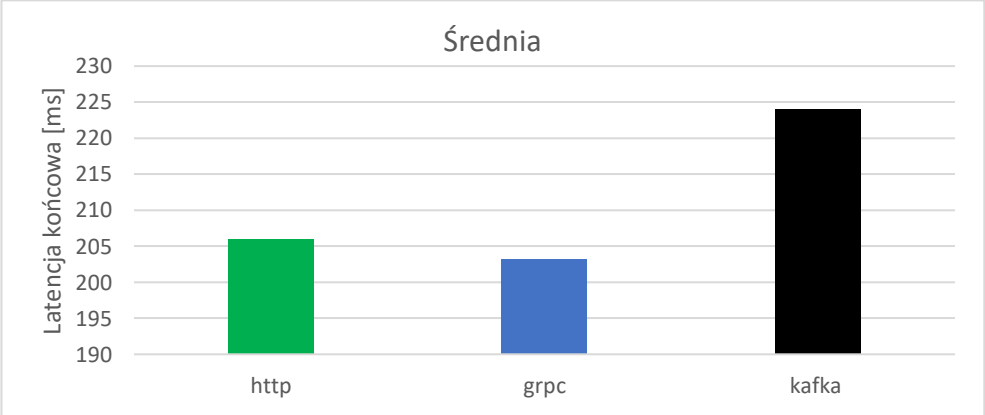


Rysunek 30. Wykres mediany latencji końcowej dla pobierania sieci społecznościowej przy małym obciążeniu (lepsza jest wartość mniejsza)

Wyniki wyglądają podobnie do pozostałych, w których celem komunikacji jest pobranie informacji. *Http* oraz *grpc* mają podobną latencję. Natomiast *kafka* ma latencję kilkukrotnie większą, ale nadal wartość 16 milisekund jest znikoma.



Rysunek 31. Wykres mediany latencji końcowej dla pobierania sieci społecznościowej przy dużym obciążeniu (lepsza jest wartość mniejsza)



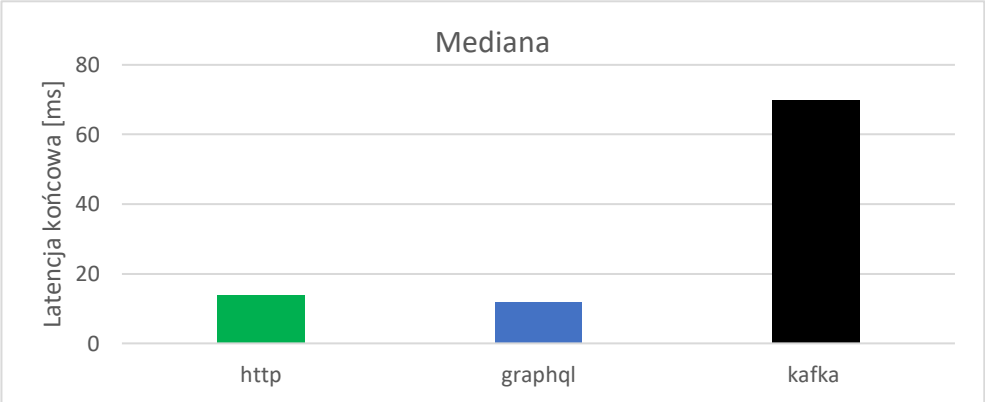
Rysunek 32. Wykres średniej latencji końcowej dla pobierania sieci społecznościowej przy dużym obciążeniu (lepsza jest wartość mniejsza)

W przypadku komunikacji pomiędzy usługami *Profile* i *Feed* skorzystano z innego rozwiązania do przekształcenia stylu komunikacji z opartej o zdarzenia na żądanie-odpowiedź. Wdrożono rozwiązanie oparte o dwie kolejki i blokowanie wątku. Dzięki zastosowaniu alternatywnej metody *kafka* była w stanie uzyskać medianę o wartości 210 ms podczas gdy mediana dla *grpc* wyniosła 190 ms a dla *http* 160 ms. Różnica jest znacznie mniejsza niż w przypadku skorzystania ze strumienia zdarzeń (*SSE*), dlatego zaleca się rozwiązanie oparte o dwie kolejki. Na rysunku 31można również zauważyć, że różnica pomiędzy *kafką* a pozostałymi technologiami zaciera się w raz z większym obciążeniem.

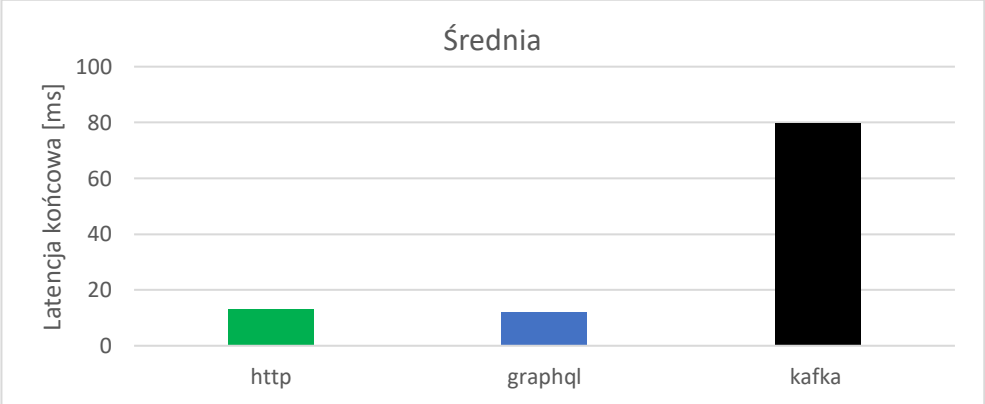
5.4.6 Pobranie postów

Ostatnią operacją braną pod uwagę jest pobranie postów wystawionych przez użytkownika. Wyniki prezentują podobne różnice do tych zaprezentowanych przy innych operacjach, które zakładają, że użytkownik oczekuje odpowiedzi od serwera. Jednak w tym przypadku oddelegowanie

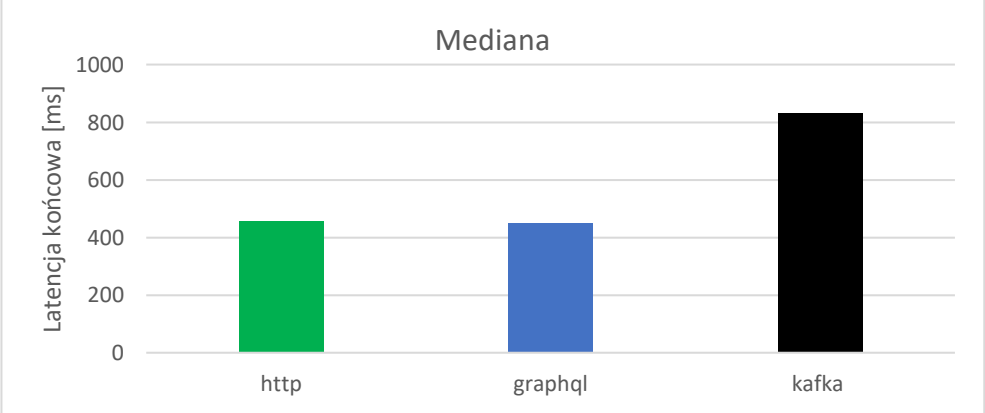
operacji za pomocą *kafki* nie przynosi żadnych korzyści z tego względu nie została uwzględniona latencja *kafki* bez odpowiedzi przesłanej strumieniem.



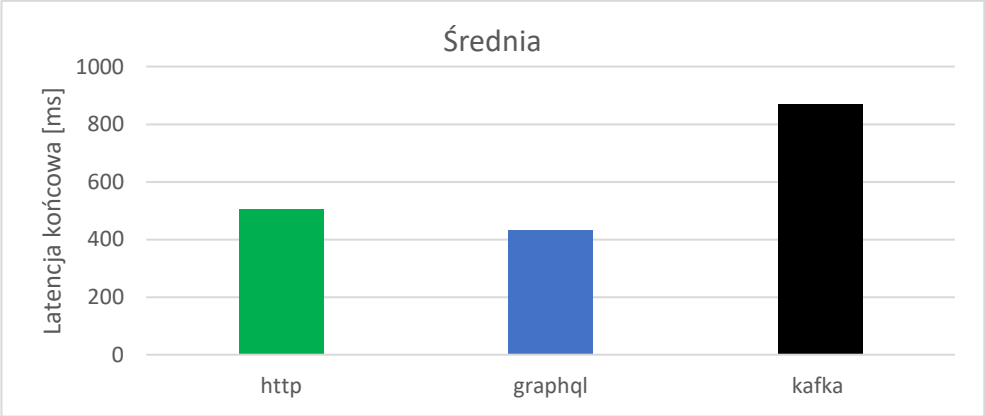
Rysunek 33. Wykres mediany latencji końcowej dla operacji pobrania postów przy małym obciążeniu (lepsza jest wartość mniejsza)



Rysunek 34. Wykres średniej latencji końcowej dla operacji pobrania postów przy małym obciążeniu (lepsza jest wartość mniejsza)



Rysunek 35. Wykres mediany latencji końcowej dla operacji pobrania postów przy dużym obciążeniu (lepsza jest wartość mniejsza)



Rysunek 36. Wykres średniej latencji końcowej dla operacji pobrania postów przy dużym obciążeniu (lepsza jest wartość mniejsza)

5.4.7 Graphql i *http*

Graphql pozwala użytkownikowi dokładnie określić jakie informacje chce uzyskać od systemu. Umożliwia to zmniejszenie czasu oczekiwania na dane oraz obciążenia systemu związanego z wielokrotnym wysłaniem zapytania.

Przykładowo użytkownik wysyła zapytanie, w którym chce uzyskać informacje o grupie, postach w niej zawartych, użytkownikach którzy utworzyli te posty oraz administratorach grupy. Używając protokołu *http* użytkownik musi wysłać 4 sekwencyjne zapytania. Istnieje możliwość wysłania równolegle zapytania o użytkowników i administratorów, co zmniejszyłoby sekwencje zapytań do 3. Dla 4 wysłanych żądań otrzymano ciąg czasów oczekiwania w milisekundach 35, 32, 32, 30. Sumarycznie oczekiwanie na dane zajęło klientowi 129 milisekund. W przypadku ograniczenia liczby żądań do 3 uzyskano 99 milisekund. Problem wysłania kilku zapytań może być rozwiązany przez kompozycje *API*, jednak podejście takie jest zbyt czasochłonne przy dużych ilościach zasobów oraz może doprowadzić do sytuacji, w której użytkownik pobiera niechciane zasoby. Rozwiązaniem tego problemu może być użycie *graphql*. W przypadku *graphql* pobranie opisanych wcześniej informacji może być zawarte w 1 żądaniu. Średni koszt takiego żądania w omawianej aplikacji to 72 milisekundy, Jest to 27 milisekund mniej niż w przypadku 3 żądań oraz 57 milisekund mniej niż w przypadku 4 żądań. Ilość żądań może być jeszcze wyższa, ponieważ nie mamy wpływu na to w jaki sposób użytkownik pobiera informacje. W skrajnym przypadku pobranie opisanych informacji może zawierać aż 6 żądań (jeżeli użytkownicy i administratorzy są pobierani osobno). Poza dłuższym oczekiwaniem klienta oraz trudniejszą w zaimplementowaniu aplikacją, wysłanie kilku żądań może znacznie bardziej obciążyć system, ponieważ wielokrotnie wykorzystany jest dostęp do bazy danych lub do usług zewnętrznych.

Podsumowując *graphql* daje szansę na uzyskanie dużo lepszej wydajności oraz zmniejszenie potencjalnej ilości zapytań. Mimo to *graphql* nie jest wolniejszym protokołem od *http*.

6. Szukanie kompromisów

Porównanie wydajności poszczególnych protokołów pokazało, że lepszą optymalność uzyskuje się nie poprzez wprowadzenie bardziej wydajnej technologii, a poprzez właściwe dopasowanie technologii do stylu komunikacji wykorzystanego w danym problemie. Z tego powodu niemożliwe jest oparcie wydajnego systemu na jednej technologii komunikacyjnej, ponieważ niewłaściwe jej użycie znacznie zmniejsza wydajność.

Jeżeli potencjalny projekt ma być prowadzony przez duży zespół oraz składać się z dziesiątek mikrousług to można dobrać technologie w następujący sposób:

- *API* wystawione dla klientów systemu najlepiej oprzeć o *graphql,* który daje następujące korzyści
    - Zmniejsza ilość pracy związaną z utrzymaniem dokumentacji
    - Upraszcza korzystanie z *API*
    - Zmniejsza potencjalną ilość zapytań
    - Daje możliwość optymalizacji złożonych zapytań
    - Umożliwia kompozycje *API* wykorzystując usługi, które implementują *graphql* lub nie
- Jeżeli któraś z usług informuje inne o zajściu jakiegoś zdarzenia lub propaguje informacje w takim przypadku najlepszym rozwiązaniem jest *kafka*, która nie obciąża serwisu emitującego zdarzenie. Jest to bardzo ważna zaleta w mikrousługach, które często korzystają z wzorców, które zakładają emitowanie zdarzeń
- Oddelegowanie długich operacji oraz implementacja operacji wykonywanych na kilku usługach (transakcje rozproszone) najlepiej wykonać z użyciem *kafki.* Dzięki trzymaniu informacji o wysłanych wiadomościach oraz możliwości wysłania wiadomości do niedostępnej usługi *kafka* jest niezawodną metodą komunikacji.
- Jeżeli zapytanie zakłada, że klient ma otrzymać odpowiedź (np. podczas pobierania informacji) należy skorzystać z *http* lub *graphql* przy komunikacji zewnętrznej. Jednak w przypadku komunikacji pomiędzy usługami najlepiej wykorzystać *grpc* lub *http.*
- Jeżeli usługi, które implementujemy nie mogą wykorzystywać protokołu *http* wtedy warto użyć *grpc* z uwagi na podobną funkcjonalność
- Jeżeli wykorzystujemy urządzenia, które nie obsługują protokołu *http* lub wymagają strumienia dwukierunkowego wtedy najlepiej użyć *grpc*, który oferuje dwukierunkowy strumień oraz pozwala statycznie określić strukturę wiadomości.

Natomiast przedstawione technologie mają też znaczące wady, które są nie do przyjęcia w określonych warunkach:

- *Kafka* z uwagi na duży próg wejścia oraz trudność w konfiguracji systemu tak aby uzyskać przyzwoitą wydajność nie jest technologią, która sprawdziłaby się w małym projekcie lub w zespole niedoświadczonym w tej technologii. *Kafka* jest najbardziej złożoną technologią z przedstawionych, która potrafi skomplikować projekt oraz uzyskać gorszą wydajność, jeżeli jest wykorzystana nieprawidłowo. Jeżeli system opiera się głównie na żądaniach typu *CRUD* (Tworzenie, Czytanie, Edycja, Usunięcie) lub zawiera małą ilość usług należy unikać *kafki,* będącej w takim przypadku niepotrzebną komplikacją. Użycie *kafki* powinno wynikać z wyraźnego powodu.
- *Graphql* jest bardzo przydatną technologią jednak jedynie w przypadku, w którym system zawiera bardzo dużą liczbę różnych zasobów. Jeżeli system nie oferuje dużej ilości zasobów,

która jest pobierana przez klientów, lub nie korzysta z kompozycji *API*, *graphql* nie oferuje żadnych korzyści.

- Protokół *http* oraz *grpc* są bardzo proste i są dobrym wyborem w małych projektach lub przy małej ilości mikrousług. Jednak potrafią być wąskim gardłem złożonego systemu, w takim wypadku zaleca się znalezienie technologii umożliwiającej poprawę wydajności

Powyższe rekomendację zostały przedstawione w tabeli 2. *N* oznacza, że danej technologii nie zaleca się, *T* wskazuje na pozytywną rekomendację, a *X* mówi o neutralności. Poszczególne cechy są posortowane priorytetem.

Tabela 2. Rekomendacje użycia danej technologii w projekcie o danych cechach

Nr	**Cecha projektu**	***http***	***grpc***	***graphql***	***kafka***
1	Komunikacja zewnętrzna	*T*	*X*	*T*	*N*
2	Komunikacja wewnętrzna	*T*	*T*	*N*	*T*
3	Operacje *CRUD* lub żądanie-odpowiedź	*T*	*T*	*T*	*N*
4	Duża liczba mikrousług	*X*	*X*	*T*	*T*
5	Mała liczba mikrousług	*X*	*X*	*N*	*N*
6	Duża ilość zasobów	*X*	*T*	*T*	*X*
7	Mała ilość zasobów	*X*	*X*	*N*	*X*
8	Duża ilość zespołów programistycznych	*X*	*T*	*X*	*T*
9	Komunikacja oparta o zdarzenia	*N*	*N*	*N*	*T*

7. Podsumowanie

Wybór technologii służącej do komunikacji jest bardzo ważną decyzja projektową. Jednak decyzja ta nie powinna wynikać z zapewnień twórców technologii związanej z jej wydajnością. Wydajny system zgodnie z uzyskanymi wynikami nie wynika z użytej technologii, a z jego zaprojektowania. Decyzja o wykorzystaniu technologii do komunikacji powinna być podjęta z uwagi na aspekty takie jak: skala projektu (ilość usług lub ilość zespołów), potencjalne obciążenie systemu, charakter informacji oraz komunikacji w poszczególnych przypadkach, oferowaną niezawodność, typ klientów korzystających z *API* oraz oczekiwania użytkowników. Natomiast najważniejszym czynnikiem jest dopasowanie stylu komunikacji do danego problemu, ponieważ problem wydajności powstaje z niepoprawnego użycia technologii. Optymalizacji systemu nie należy zaczynać od zmiany wykorzystanej technologii, a od weryfikacji architektury aplikacji, implementacji odpowiednich wzorców architektonicznych oraz usuwaniu błędów popełnionych przy projektowaniu systemu.

Każda z omówionych technologii posiadała wady i zalety. Najważniejsze z nich zostały wylistowane poniżej:

- *Kafka*
    - Zalety
        - Znikomy czas zablokowania klienta
        - Utrwalona historia wysłanych wiadomości
        - Brak sprzężenia pomiędzy usługami
        - Niezawodność
        - Technologia jest wspierana
        - Oferowane dodatkowe funkcjonalności takie jak *Transacions API, Stream AP*I lub *Connectros*
        - Możliwość skalowania każdej operacji z osobna
        - Możliwość konfiguracji wielu aspektów takich jak partycje, *batch, ack* itp.
    - Wady
        - Brak możliwości wydajnej obsługi zapytań pobierających informacje
        - Wysoki próg wejścia dla programistów
        - Wymaga dodatkowej infrastruktury
        - Z powodu bezpieczeństwa nie zaleca się użycia jej do komunikacji zewnętrznej
        - Duża ilość konfiguracji potrzebnej do dostosowania w celu uzyskania przyzwoitej latencji końcowej
        - Semantyka „dokładnie raz” możliwa jedynie po wprowadzeniu dodatkowych mechanizmów
        - Wprowadza problem *Eventual Consistency*
- *Http*
    - Zalety
        - Bardzo dobrze znany przez większość programistów
        - Od 2023 używa *UDP*, co wprowadzi większą wydajność
        - Technologia jest wspierana
        - Niski próg wejścia dla developerów
        - Standardowa technologia używana w każdej przeglądarce

    - Wady
        - W praktyce sam protokół nie wystarcza i trzeba wprowadzić *REST* oraz *OpenAPI*
        - Brak możliwość określenia wymaganych przez klienta informacji
        - W językach bez statycznego typowania istnieje potrzeba wprowadzenia walidacji schematu wiadomości
        - Duża liczba zasobów systemu wprowadza dużą liczbę punktów końcowych
        - Optymalizacja sekwencji zapytań jest po stronie klienta
        - Wprowadza sprzężenie usług
        - Format *json* wymaga określenia kontraktu pomiędzy usługami tworzonymi przez oddzielne zespoły
- *Grpc*
    - Zalety
        - Możliwość określenia schematu wiadomości
        - Najniższy próg wejścia dla developerów
        - Technologia jest wspierana
        - Możliwość wygenerowania biblioteki zawierającej gotowe projekcje (kontrakt)
        - Daje możliwość skorzystania z dwukierunkowego strumienia danych
        - Twórcy obiecali oparcie przyszłej wersji protokołu o *http3*
    - Wady
        - Podczas testów wydajnościowych nie udało się uzyskać lepszej wydajności niż w przypadku *http*
        - Brak możliwość określenia wymaganych przez klienta informacji
        - Optymalizacja sekwencji zapytań jest po stronie klienta
        - Wprowadza sprzężenie usług
        - Zwykle wymaga wprowadzenie dodatkowego procesu CI/CD generującego biblioteki z projekcjami
- *Graphql*
    - Zalety
        - Możliwość określenia potrzebnych informacji przez klienta
        - Wsparcie w postaci federacji podczas tworzenia kompozycji
        - Możliwość wprowadzenia optymalizacji w przypadku zagnieżdżonych zapytań
        - Upraszcza komunikacje ze złożonymi *API*
        - Umożliwia określenie schematu wiadomości
        - Zmniejsza potencjalną ilość zapytań
    - Wady
        - Wprowadza dodatkowy obowiązek zaprojektowania grafu zasobów
        - Mniejsza ilość dostępnych bibliotek w przypadku technologii innych niż JavaScript
        - Brak wprowadzenie blokady zbyt złożonych zapytań oraz brak optymalizacji problemu *N+1* skutkuje dużym spadkiem wydajności
        - W przypadku komunikacji pomiędzy usługami nie oferuje żadnych korzyści

Prace cytowane

[1]. *IBM, IBM Microservices in the enterprise* https://www.ibm.com/downloads/cas/OQG4AJAM, Dostęp 31 05 2023

[2]. *JetBrains, The State of Developer Ecosystem 2021,* https://www.jetbrains.com/lp/devecosystem-2021/, Dostęp 31 05 2023

[3]. S. Newman, *Building Microservices: Designing Fine-Grained Systems* ISBN: 9781491950319, *O'Reilly* , 2015.

[4]. E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software* ISBN: 9780132181273, *Wesley Professional*, 2003.

[5]. R. C. Martin, *Clean Code* ISBN: 978-0132350884, *Pearson*, 2008.

[6]. R. C. Martin, *Clean Architecture* ISBN: 978-0134494166, *Pearson*, 2017.

[7]. C. Richardson, *Microservices Patterns* ISBN: 978-1617294549, *Manning*, 2018.

[8]. *JetBrains, Kotlin,* https://kotlinlang.org/, Dostęp 31 05 2023

[9]. *Gradle Inc., Gradle*, https://gradle.org/, Dostęp 31 05 2023

[10]. *VMware Tanzu, Spring*, https://spring.io/, Dostęp 31 05 2023

[11]. *MongoDB Inc., MongoDB*, https://www.mongodb.com/, Dostęp 31 05 2023

[12]. *Docker Inc., Docker*, https://www.docker.com/, Dostęp 31 05 2023

[13]. *Docker Inc., Docker Compose*, https://docs.docker.com/compose/, Dostęp 31 05 2023

[14]. *RedHat, KeyCloak*, https://www.keycloak.org/, Dostęp 31 05 2023

[15]. *Postman Inc., Postman*, https://www.postman.com/, Dostęp 31 05 2023

[16]. *Python Software Foundation, Python*, https://www.python.org/, Dostęp 31 05 2023

[17]. *Apache, Jmeter*, https://jmeter.apache.org/, Dostęp 31 05 2023

[18]. Hgraca, *DDD, Hexagonal, Onion, Clean, CQRS, … How I put it all together* 5 06 2023. https://herbertograca.com/2017/11/16/explicitarchitecture01-ddd-hexagonal-onion-cleancqrs-how-i-put-it-all-together/, Dostęp 31 05 2023

[19]. C. Walls, *Spring w akcji Wydanie V* ISBN: 978-83-283-5606-1, *Helion* , 2019.

[20]. *Mozilia Corporations, An overview of HTTP*, https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview, Dostęp 31 05 2023

[21]. *W3Techs, Statystyki dotyczące aplikacji internetowych*, https://w3techs.com/technologies/overview/site_element, Dostęp 31 05 2023

[22]. *JetBrains, The State Of Developer Ecosystem 2022*, https://www.jetbrains.com/lp/devecosystem-2022/microservices/, Dostęp 31 05 2023

[23]. *IBM, What is a REST API?*, https://www.ibm.com/topics/rest-apis, Dostęp 31 05 2023

[24]. *G. Foundation, GraphQL* https://graphql.org/ Dostęp 31 05 2023

[25]. *Apache, Kafka*, https://kafka.apache.org/, Dostęp 31 05 2023

[26]. *Apache, Zookeeper*, https://zookeeper.apache.org/, Dostęp 31 05 2023

[27]. N. Narkhede, *Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale* ISBN: 9781491936160, *O'Reilly* , 2017.

[28]. *C. N. C. Foundation, GRPC* https://grpc.io/, Dostęp 31 05 2023

[29]. M. Richards, *Fundamentals of Software Architecture* ISBN: 978-1492043454, *O'Reilly* , 2020.

[30]. *Wikipedia, Two Generals' Problem*, https://en.wikipedia.org/wiki/Two_Generals%27_Problem, Dostęp 31 05 2023

[31]. *OpenAPI Initiative, OpenAPI-Specification*, https://github.com/OAI/OpenAPI-Specification/blob/main/versions/3.0.2.md, Dostęp 31 05 2023

[32]. Mozilla Corporation', *server-sent events*, https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events, Dostęp 31 05 2023

Spis ilustracji

Spis listingów

Spis tabel