

Obliczenia rozproszone w klastrach i gridach

Temat:

*Prometheus jako narzędzie
monitorowania systemów rozproszonych*

Zespół:

- Karol Adameczek nr indeksu 299231
- Bartosz Michalak nr indeksu 329127
- Norbert Piecka nr indeksu 329131

Przykładowy projekt prezentujący implementację systemu Prometheus:

- <https://github.com/NorbertPiecka/ORWKiG>

Spis treści

1. Wstęp	3
1.1. Cel i znaczenie monitorowania systemów rozproszonych	3
1.2. Wyzwania związane z monitorowaniem w środowiskach rozproszonych	4
2. Prometheus jako narzędzie do monitorowania systemów rozproszonych	6
2.1. Rola Prometheusa jako narzędzia monitorowania.....	6
2.2. Architektura Prometheusa.....	7
3. Opis funkcjonalności Prometheus	9
3.1. Konfiguracja podstawowa Prometheusa	9
3.2. Zabezpieczanie Prometheusa.....	10
3.3. Narzędzia wspierające	11
3.4. Wspierane języki programowania	13
3.5. Porównanie Prometheus z konkurencją	13
4. Przykładowe zastosowanie w projekcie	15
4.1. Uruchomienie systemu Prometheus z wykorzystaniem kontenera Docker	15
4.2. Połączenie do Prometheus z wykorzystaniem języków programowania Java i Python.....	16
4.2.1. Java.....	17
4.2.2. Python	19
4.3. Przegląd metryk w Prometheusie.....	20
5. Podsumowanie i wnioski	22
6. Bibliografia	23

1. Wstęp

Tematem pracy jest wykorzystania rozwiązania systemu Prometheus jako narzędzia monitorowania systemów rozproszonych. Warto zatem wspomnieć czym są systemy rozproszone [1]:

“System rozproszony to zbiór programów komputerowych, które wykorzystują zasoby obliczeniowe w wielu oddzielnych węzłach obliczeniowych, aby osiągnąć wspólny cel. Systemy rozproszone mają za zadanie eliminować z systemu wąskie gardła lub centralne punkty awarii.”

Jak wskazuje definicja, stosowanie systemu rozproszonego jest rozwiązaniem na część problemów, które są tym bardziej uporczywe, im rozwiązanie, nad którym pracujemy, staje się większe i bardziej złożone. Jednakże wiąże się to także z wadami - główną z nich jest zdecydowanie większa trudność zarządzania infrastrukturą i poszczególnymi programami składowymi, które mogą być rozproszone nawet na wiele miast, krajów czy nawet kontynentów. Z tego powodu na rynku powstały narzędzia monitorowania systemów rozproszonych, które po poprawnej konfiguracji, znacząco ułatwiają pracę administratora systemu rozproszonego.

W pracy tej skupimy się na opisanu jednego z konkretnych rozwiązań, które pozwala na monitorowanie takich systemów - Prometheus. Na początek przedstawione zostaną ogólne cele i wyzwania z jakimi mierzy się to narzędzie. Następnie przedstawimy w jaki sposób zostało zaprojektowane, aby tym celom i wyzwaniom sprostać. W kolejnym rozdziale bliżej pokazany będzie sposób pracy z Prometheus, konkretne funkcjonalności, integrację z innymi rozwiązaniami, a także porównanie do innych podobnych istniejących narzędzi. Później pokazane będzie praktyczne zastosowanie tego narzędzia monitorowania systemu w projekcie. Na końcu znajduje się podsumowanie treści i wnioski wyciągnięte na temat Prometheusa.

1.1. Cel i znaczenie monitorowania systemów rozproszonych

Główną przyczyną implementacji monitorowania w systemach rozproszonych jest zapewnienie stabilności, niezawodności i wydajności. Jest to proces, który polega na ciągłym zbieraniu, analizowaniu i raportowaniu danych dotyczących działania systemu. Jednym z głównych celów jest wczesne wykrywanie problemów i identyfikację zagrożeń takich jak awarie, opóźnienia, wysokie obciążenie czy błędy. Dzięki temu można podjąć odpowiednie działania naprawcze w krótkim czasie, aby zapobiec wystąpieniu negatywnych zjawisk mających wpływ na komfort lub samą możliwość korzystania z systemu przez użytkownika.

Poza wspomaganiem wcześniejszej reakcji, monitorowanie pozwala na optymalizację wydajności utrzymywanego systemu. Śledzenie wydajności poszczególnych elementów infrastruktury, takich jak serwery, bazy danych czy sieć umożliwia zidentyfikowanie obszarów, które wymagają optymalizacji. Pozwala to podjąć odpowiednie działania, mające na celu poprawę wydajności i szybkości działania systemu. Dodatkowo warto wspomnieć, że dzięki analizie danych zebranych podczas procesu monitorowania systemu, jesteśmy w stanie zweryfikować i udoskonalić procesy biznesowe, które realizuje system. Takie informacje jak wydajność poszczególnych obszarów systemu, pozwalają na zidentyfikowanie potencjalnych elementów systemu, które powinny zostać usprawnione w celu poprawy efektywności i jakości działania całego rozwiązania.

Kolejnym celem implementacji monitorowania w systemie jest zapewnienie jego bezpieczeństwa. Zbieranie informacji o systemie w czasie rzeczywistym pozwala na wykrywanie i reagowanie na potencjalne zagrożenia i ataki cybernetyczne. Większość istniejących na rynku rozwiązań pozwala na zdefiniowanie powiadomień (ang. alerts), które otrzymuje administrator systemu w momencie spełnienia wcześniej zdefiniowanego warunku. Takie reguły opierają się na logach aplikacji, które są przesyłane i analizowane w rozwiązaniu monitorującym system. Korzystając z tego rozwiązania oraz monitorowanie ruchu sieciowego można wykryć podejrzaną aktywność i podjąć działania mające na celu ochronę systemu przed atakami.

Jednym z bardzo praktycznych zastosowań monitorowania systemów rozproszonych jest umożliwienie analizy obciążenia i wykorzystania zasobów. Na podstawie zebranych danych można dokładnie określić, jakie zasoby są potrzebne do utrzymania optymalnej wydajności systemu i odpowiednio planować skalowanie infrastruktury potrzebnej do działania systemu.

Biorąc pod uwagę wszystkie wyżej wymienione cele, monitorowanie systemów rozproszonych przyczynia się do poprawy jakości obsługi użytkowników, minimalizacji czasu niedostępności w przypadku awarii i zmniejszenia ryzyka jej wystąpienia oraz zwiększenia wydajności oraz bezpieczeństwa systemu, a także optymalizacji kosztów i zasobów potrzebnych na poprawne działanie systemu. Rozwiązania monitorujące są niezbędne w przypadku pracy z dużymi systemami rozproszonymi. Bez nich utrzymanie kontroli nad działaniem systemu jest praktycznie niemożliwe, a na pewno znacznie utrudnione.

1.2. Wyzwania związane z monitorowaniem w środowiskach rozproszonych

Monitorowanie systemów należy do wymagających zadań, szczególnie jeśli mamy do czynienia z systemami rozproszonymi. Z tym procesem wiążą się wyzwania związane ze złożonością i skalą systemu, który monitorujemy.

Jednym z nich jest skalowalność rozwiązania. Systemy rozproszone zazwyczaj składają się z wielu komponentów, które mogą być rozmieszczone na wielu serwerach, centrach danych, znajdujących się w różnych lokalizacjach geograficznych. Monitorowanie wszystkich tych komponentów i zarządzanie produkowanymi przez nie danymi na dużą skalę może być poważnym wyzwaniem.

Następnym problemem, który występuje podczas monitorowania systemów w środowiskach rozproszonych jest zbieranie danych. Różne komponenty mogą generować dzienniki, metryki lub zdarzenia w różnych formatach, co utrudnia skuteczną konsolidację i analizę danych. W tym przypadku wymagane jest wsparcie dla wielu różnych dostępnych na rynku formatów danych oraz sposób ich standaryzacji do jednego oficjalnego formatu dla danego rozwiązania monitorującego system, aby jak najbardziej ułatwić pracę użytkownika z systemem.

Systemy rozproszone często działają w wielu sieciach i polegają na złożonych topologiach sieci. Zarządzanie nimi nie należy do łatwych zadań. W celu zaimplementowania poprawnego systemu monitorowania takich systemów wymagane jest zrozumienie podstawowej infrastruktury sieciowej i zapewnienia odpowiedniego wglądu w ruch sieciowy i komunikację między komponentami.

Kolejnym problemem jest zapewnienie dynamizmu i elastyczności rozwiązania monitorującego. Systemy rozproszone zazwyczaj są zaprojektowane do dynamicznego skalowania w zależności od zapotrzebowania klienta. Ta dynamika wprowadza wyzwania w monitorowaniu, ponieważ liczba komponentów może się szybko zmieniać, a system musi być gotowy na wykrycie i obsługę zmian wprowadzanych w systemie. Kluczowe w tym procesie jest przygotowanie odpowiedniej konfiguracji narzędzia monitorującego w celu dostosowania się do zmieniającego się stanu systemu.

W systemach rozproszonych często stosowane są techniki replikacji i partycjonowania danych w celu osiągnięcia odporności na uszkodzenia systemu oraz poprawę wydajności. Wymaga to zapewnienia spójności danych, która jest szczególnie wymagana jeśli chcemy korzystać z rozwiązań monitorujących. Dane muszą dokładnie odzwierciedlać stan, w którym znajduje się system. W przeciwnym wypadku będą następować przekłamania, a sam wynik analizy danych dostarczonych do rozwiązania monitorującego będzie niepewny. Powoduje to utrudnienie podjęcia działania w celu poprawy jakości systemu.

Systemy mogą ulegać awariom, a rozwiązania monitorujące muszą być przygotowane na obsługę takich scenariuszy. W przypadku wystąpienia błędu powinny zostać uruchomione odpowiednie mechanizmy odpowiedzialne za zapewnienie odporności na błędne działanie systemu lub niespodziewane zachowania takie jak nadmiarowość danych przychodząca z systemu przekraczająca możliwości procesowania danych przez system monitorujący. Innym przypadkiem jest zapewnienie przełączania awaryjnego lub poinformowania o utracie połączenia z systemem w przypadku awarii oraz możliwość odzyskania danych aplikacji na podstawie logów. Narzędzia do monitorowania powinny być w stanie płynnie radzić sobie z problemami, które mogą wystąpić w systemie i zapewniać dokładny wgląd w stan i wydajność, nawet podczas scenariuszy awarii.

W systemach rozproszonych występuje wiele zależności między komponentami, a monitorowanie jednego z nich w izolacji może nie zapewniać pełnego obrazu zachowania systemu. Zrozumienie i wizualizacja zależności między komponentami ma kluczowe znaczenie dla dokładnego zidentyfikowania tak zwanych wąskich gardeł, problemów z wydajnością lub awarii występujących w systemie.

Kolejnym wyzwaniem jest analiza korelacji zdarzeń i głównych przyczyn. W systemach rozproszonych identyfikacja głównej przyczyny problemu może być trudna ze względu na wcześniej wspomniane złożone interakcje między komponentami. Korelowanie zdarzeń i metryk z różnych komponentów oraz określenie potencjalnej przyczyny problemu wymaga zaawansowanych technik monitorowania i analizy. Nowoczesne systemy monitorujące posiadają taką opcję.

Bardzo ważną cechą systemu jest jego bezpieczeństwo i zapewnienie prywatności. Monitorowanie systemów rozproszonych może budzić obawy, ponieważ w grę mogą wchodzić wrażliwe dane i informacje na temat komunikacji komponentów systemu. Narzędzia do monitorowania powinny zapewniać bezpieczną transmisję danych, kontrolę dostępu oraz mechanizmy anonimizacji danych w celu ochrony wrażliwych informacji.

Ostatnim poruszonym w tej pracy wyzwaniem jest złożoność operacyjna związana z wdrażaniem i utrzymywaniem infrastruktury monitorowania dla systemów rozproszonych. Ten proces może okazać się skomplikowany. Wymagane jest między innymi poprawna konfiguracja agentów monitorowania, konfiguracja reguł dotyczących zbierania danych, tworzenie i zarządzanie alertami oraz powiadomieniami, a także zapewnienie zasięgu w całym systemie. Te wszystkie operacje wymagają starannego planowania i ciągłej konserwacji rozwiązania monitorującego.

Sprostanie tym wyzwaniom wymaga połączenia odpowiednich narzędzi monitorowania, skutecznych technik agregacji wraz z analizą danych oraz głębokiego zrozumienia architektury, a także znakomita znajomość zachowania systemu rozproszonego.

2. Prometheus jako narzędzie do monitorowania systemów rozproszonych

Jednym z rozwiązań, które można wykorzystać w celu monitorowania systemów rozproszonych jest system Prometheus. Jest to zaawansowane narzędzie oferujące szereg przydatnych funkcji wspierających proces monitorowania zachowania systemu. Poniżej zostaną przedstawione kluczowe rozwiązania wspierane przez Prometheus, które pozwalają rozwiązać wyżej zdefiniowane wymagania, które stają przed systemem monitorującym.

2.1. Rola Prometheusa jako narzędzia monitorowania

Jednym z poruszanych wyżej aspektów związanych z systemami monitorowania jest zbieranie danych na temat funkcjonowania systemu. Prometheus oferuje opcję gromadzenia i przechowania danych indeksowanych za pomocą czasu dotarcia danych z różnych komponentów systemu rozproszonego, takich jak serwery, aplikacje i usługi. Wykorzystuje model oparty na okresowym pobieraniu metryk i danych ze skonfigurowanych komponentów za pomocą protokołu HTTP[2]. Taki proces gromadzenia danych zapewnia wgląd w kondycję, wydajność i zachowanie systemu rozproszonego w czasie rzeczywistym.

Do przechowywania metryk Prometheus wykorzystuje własną bazę danych szeregów czasowych[3]. Zapewnia ona wydajną współpracę z zebranymi danymi oraz metrykami. Baza danych jest zoptymalizowana pod kątem wysokiej przepustowości zapisu i zapytań oraz obsługuje elastyczne możliwości indeksowania i wykonywania zapytań. Ten mechanizm przechowywania umożliwia przechowywanie i analizowanie danych historycznych, umożliwiając analizę trendów, rozwiązywanie problemów i optymalizację wydajności systemu.

Wyszukiwanie i analiza danych są zapewnione dzięki autorskiemu językowi zapytań o nazwie PromQL (Prometheus Query Language)[4]. Umożliwia on użytkownikom tworzenie złożonych zapytań i pobieranie określonych metryk lub agregatów na podstawie różnych wymiarów i etykiet. PromQL wspiera analizę ad-hoc, wykrywanie anomalii i korelację wskaźników. Ułatwia to znacząco analizę przyczyn źródłowych i proaktywne monitorowanie systemów rozproszonych.

Jeśli chodzi o powiadomienia na temat stanu systemu, rozwiązanie Prometheus zawiera solidny system alertów, który umożliwia konfigurację alertów na podstawie predefiniowanych reguł lub warunków[5]. Administrator ma możliwość zdefiniowania reguły alertu za pomocą wyrażeń w języku PromQL. Następnie określone są kanały powiadomień. Do wspieranych aktualnie należą np. e-mail, Slack, PagerDuty[6]. Alert zostaje wysłany, gdy zostaną spełnione określone progi lub warunki zdefiniowane przez użytkownika. Ta funkcja należy do bardzo popularnych w systemach monitorowania i umożliwia natychmiastowe powiadamianie operatorów o wszelkich problemach lub nietypowym zachowaniu w systemie rozproszonym.

Następnym aspektem zapewniającym ułatwienie w interpretowaniu stanu systemu jest opcja wizualizacji i pulpity nawigacyjne. Chociaż sam Prometheus koncentruje się na gromadzeniu i przechowywaniu danych, posiada opcję łatwej integracji z narzędziami do wizualizacji, takimi jak Grafana[7]. Te rozwiązanie zapewnia bogaty zestaw opcji wizualizacji danych, umożliwiając użytkownikom tworzenie interaktywnych pulpitów nawigacyjnych i wizualnych reprezentacji monitorowanych metryk. Te pulpity nawigacyjne pomagają operatorom oraz interesariuszom uzyskać wgląd w wydajność i trendy systemu, ułatwiając procesy monitorowania i podejmowania decyzji.

System Prometheus dysponuje szeroką gamą eksporterów[8] i bibliotek[9], które umożliwiają wsparcie w integrowaniu rozwiązania z różnymi komponentami systemów rozproszonych i platformami. Ułatwia to zbieranie metryk wykorzystując popularne technologie, takich jak Kubernetes, Apache Kafka, Redis i inne. Prometheus obejmuje również szerokie wsparcie ze strony aktywnie działającej społeczności, która rozszerza te narzędzie monitorowania poprzez dodanie wsparcia dla komponentów systemów rozproszonych, a także wzbogaca bazę wiedzy odnośnie najlepszych praktyk i zapewnia ciągły rozwój i wsparcie. Takie zaangażowanie zapewnia, że narzędzie Prometheus pozostaje solidnym i niezawodnym rozwiązaniem do monitorowania systemów rozproszonych.

Jeżeli chodzi o skalowalność i rozszerzalność to Prometheus został zaprojektowany tak, aby był wysoce skalowalny i mógł obsługiwać rozproszone systemy na dużą skalę. Obsługuje konfiguracje federacyjne[10], umożliwiając agregację wielu instancji Prometheus i wysyłanie zapytań jako pojedynczy system logiczny. Ponadto system Prometheus oferuje interfejs API do niestandardowych integracji. Umożliwia to rozszerzalność i dostosowanie się do unikalnych wymagań monitorowania dla niestandardowych aplikacji.

Podsumowując, narzędzie Prometheus może być traktowane jako pełnoprawne centralne rozwiązanie monitorujące system. Umożliwia gromadzenie danych oraz ich przechowywanie, wysyłanie zapytań, alerty, wizualizację, a także wsparcie podczas integracji z systemami rozproszonymi. Jego funkcje i możliwości sprawiają, że jest to popularny wybór do monitorowania kondycji, wydajności i niezawodności złożonych infrastruktur rozproszonych.

2.2. Architektura Prometheusa

Architekturę Prometheusa jest złożona, ale dzięki podziałowi odpowiedzialności jest bardzo intuicyjna. Można ją podzielić na kilka kluczowych komponentów wraz z wyszczególnionymi interakcjami pomiędzy nimi[11].

Pierwszym omawianym jest Serwer Prometheus, który stanowi rdzeń architektury. Odpowiada za gromadzenie, przechowywanie, przetwarzanie, wysyłanie zapytań i alarmowanie. Serwer okresowo pobiera dane metryk ze skonfigurowanych elementów systemu rozproszonego, a następnie przechowuje zebrane dane w swojej bazie danych szeregów czasowych. Umożliwia to w dalszej części ocenę reguły alertów, na podstawie której, wysyłane są później alerty. Dodatkowo Serwer Prometheus udostępnia interfejs API z wykorzystaniem protokołu HTTP[12] do pobierania danych i konfiguracji monitorowania.

Następnym komponentem jest ten odpowiedzialny za zbieranie metryk. Prometheus zbiera dane metryk przy użyciu modelu opartego na ściąganiu. Działa jak klient HTTP, okresowo wysyłając zapytania do punktów końcowych, z których pozyskiwane są metryki – docelowych systemów lub

usług (np. aplikacji, serwerów lub eksporterów). Cele mogą ujawniać metryki przy użyciu formatu ekspozycji Prometheus, który obejmuje proste formaty tekstowe, takie jak zwykły tekst, lub bardziej wydajne formaty, takie jak Protocol Buffers[13]. Prometheus używa tego modelu do zbierania danych z komponentów systemu rozproszonego.

Do przechowywania danych metryk system Prometheus wykorzystuje wbudowaną bazę danych szeregów czasowych[3]. Jest ona zoptymalizowana pod kątem wysokiej przepustowości zapisu i wydajnego wykonywania zapytań. Dane są w niej organizowane na podstawie kombinacji nazw metryk - par klucz i wartość zwanych etykietami oraz sygnatury czasowej. Ten model danych umożliwia elastyczne formułowanie i wykonywanie zapytań oraz wydajne przechowywanie danych szeregów czasowych.

Zapytania i przetwarzanie danych w systemie Prometheus są realizowane przy użyciu potężnego języka zapytań o nazwie PromQL (Prometheus Query Language)[4]. PromQL umożliwia użytkownikom wyrażanie złożonych zapytań i pobieranie określonych metryk lub agregacji w oparciu o różne wymiary i etykiety. Sformułowane zapytania są oceniane przez Serwer Prometheus i przetwarzane na podstawie przechowywanych danych szeregów czasowych w bazie danych, zwracając wynik lub wyniki użytkownikowi.

Za obsługę powiadomień w systemie Prometheus odpowiada podsystem alertów, który umożliwia użytkownikom definiowanie reguł na podstawie określonych warunków lub progów[5]. Reguły te są wyrażone w języku PromQL i są stale oceniane przez serwer. Po spełnieniu warunku Prometheus uruchamia alert z nim związany i wysyła powiadomienia do skonfigurowanych kanałów np. e-mail. Umożliwia to sprawne powiadamianie operatorów lub interesariuszy o potencjalnych problemach lub nietypowym zachowaniu systemu.

System Prometheus wspiera również wykrywanie nowych usług. Obsługuje różne mechanizmy wykrywania usług w celu dynamicznego dostosowania się i monitorowania nowych celów będących częścią systemu rozproszonego[6]. Może wykorzystywać statyczne pliki konfiguracji komponentów systemu, wykrywanie usług oparte na DNS lub opierać się na integracji z platformami orkiestracji, takimi jak Kubernetes. Pozwala to Prometheusowi dostosować się do dynamicznych środowisk, w których liczba i lokalizacja komponentów systemu może być zmienna w czasie.

Prometheus udostępnia biblioteki klienckie[9] i eksportery[8], które upraszczają proces oprzyrządowania aplikacji i systemów do gromadzenia danych metrycznych. Eksporterzy działają jako pomosty między Prometheusem a różnymi systemami lub komponentami stron trzecich, umożliwiając mu pobieranie z nich danych. Te biblioteki i eksportery umożliwiają łatwą integrację z popularnymi technologiami i frameworkami aktualnie używanymi na rynku.

Chociaż Prometheus koncentruje się na gromadzeniu i przechowywaniu danych, jest powszechnie używany w połączeniu z popularnym narzędziem do wizualizacji Grafana[7]. Zastosowanie takiej konfiguracji pozwala na wyszukiwanie i wizualizowanie zebranych danych metryk. Grafana zapewnia elastyczny i konfigurowalny interfejs pulpitu nawigacyjnego przystosowanego do wizualizacji metryk, tworzenia wykresów oraz generowania wizualnych reprezentacji monitorowanych danych.

Architektura rozwiązania Prometheus została zaprojektowana i przygotowana do zapewnienia skalowalnego i elastycznego rozwiązania monitorowania systemów rozproszonych. Umożliwia wydajne gromadzenie, przechowywanie, wysyłanie zapytań, ostrzeganie i wizualizację danych. Zapewnia to administratorom systemu uzyskanie wglądu w stan, wydajność i zachowanie systemu.

3. Opis funkcjonalności Prometheus

Prometheus jako narzędzie do monitorowania systemów rozproszonych, a co za tym idzie systemów, które mogą być napisane w różnych językach programowania oraz wymagają spersonalizowanych rozwiązań pozwalających na wysyłanie metryk musi wspierać wiele języków programowania oraz wyróżniać się na tle konkurencji. Poniżej opisana została konfiguracja podstawowa Prometheusa, narzędzia rozszerzające jego funkcjonalności, wspierane języki programowania, a również wykonane zostało porównanie z konkurencją.

3.1. Konfiguracja podstawowa Prometheusa

Konfiguracja Prometheusa może odbywać się na dwa sposoby[6]:

- Ustawienie odpowiednich flag w poleceniu linii komend
- Plik konfiguracyjny `prometheus.yml`

Konfiguracja za pomocą linii komend pozwala na ustawienie takich wartości jak lokalizacja folderu na metryki, ilość danych trzymany na dysku oraz w pamięci, ale także wskazanie lokalizacji pliku konfiguracyjnego czy zmiana poziomu logowania. Oczywiście nie są to wszystkie możliwości linii komend, więc opcji może zostać sprawdzonych poprzez wpisanie komendy: `/prometheus -h`, wywołanie tej komendy zostało przedstawione na Rys.1. Wprowadzenie zmian w konfiguracji na poziomie linii komend wymaga wyłączenia aplikacji i uruchomienia jej jeszcze raz z nowymi parametrami.

```
The Prometheus monitoring server

Flags:
-h, --[no-]help            Show context-sensitive help (also try
                           --help-long and --help-man).
--[no-]version             Show application version.
--config.file="prometheus.yml"
                           Prometheus configuration file path.
--web.listen-address="0.0.0.0:9090"
                           Address to listen on for UI, API, and
                           telemetry.
--web.config.file=""        [EXPERIMENTAL] Path to configuration file that
                           can enable TLS or authentication.
--web.read-timeout=5m       Maximum duration before timing out read of the
                           request, and closing idle connections.
--web.max-connections=512   Maximum number of simultaneous connections.
--web.external-url=<URL>    The URL under which Prometheus is externally
                           reachable (for example, if Prometheus is served
                           via a reverse proxy). Used for generating
                           relative and absolute links back to Prometheus
                           itself. If the URL has a path portion, it will
                           be used to prefix all HTTP endpoints served by
                           Prometheus. If omitted, relevant URL components
                           will be derived automatically.
--web.route-prefix=<path>   Prefix for the internal routes of
                           web endpoints. Defaults to path of
                           --web.external-url.
--web.user-assets=<path>    Path to static asset directory, available at
                           /user.
--[no-]web.enable-lifecycle
                           Enable shutdown and reload via HTTP request.
--[no-]web.enable-admin-api
                           Enable API endpoints for admin control actions.
--[no-]web.enable-remote-write-receiver
                           Enable API endpoint accepting remote write
                           requests.
--web.console.templates="consoles"
                           Path to the console template directory,
                           available at /consoles.
--web.console.libraries="console_libraries"
                           Path to the console library directory.
--web.page-title="Prometheus Time Series Collection and Processing Server"
```

Rys. 1. Wywołanie komendy: `/prometheus -h`.

Źródło: Rysunek autorski.

Drugim sposobem konfiguracji Prometheusa jest wykorzystanie pliku konfiguracyjnego `prometheus.yml`. Przed jego stworzeniem należy się upewnić, że została ustawiona odpowiednia flaga `--config.file`, widoczna na Rys.3.1. Plik ten pozwala na zdefiniowanie wszystkich ustawień dotyczących serwisów zbierających metryki oraz ich instancji. Zaletą tego typu konfiguracji jest przede wszystkim możliwość jej zmiany w trakcie działania aplikacji i wdrożenia bez konieczności restartu. Dzieje się tak poprzez przesłanie sygnału `SIGHUP` do Prometheusa lub poprzez wysłanie zapytania `POST` na odpowiedni endpoint, wymaga to jednak wcześniej włączenia specjalnej flagi `--web.enable-lifecycle`. Plik konfiguracyjny jest zapisany w formacie `YAML` oraz posiada zdefiniowany przez Prometheusa schemat. Podstawowy plik konfiguracyjny został przedstawiony na Rys.2.

```
1  # my global config
2  global:
3    scrape_interval: 15s # Set the scrape interval to every 15 seconds. Default is every 1 minute.
4    evaluation_interval: 15s # Evaluate rules every 15 seconds. The default is every 1 minute.
5    # scrape_timeout is set to the global default (10s).
6
7  # Alertmanager configuration
8  alerting:
9    alertmanagers:
10     - static_configs:
11       - targets:
12         # - alertmanager:9093
13
14  # Load rules once and periodically evaluate them according to the global 'evaluation_interval'.
15  rule_files:
16    # - "first_rules.yml"
17    # - "second_rules.yml"
18
19  # A scrape configuration containing exactly one endpoint to scrape:
20  # Here it's Prometheus itself.
21  scrape_configs:
22    # The job name is added as a label `job=<job_name>` to any timeseries scraped from this config.
23    - job_name: "prometheus"
24
25      # metrics_path defaults to '/metrics'
26      # scheme defaults to 'http'.
27
28      static_configs:
29        - targets: ["localhost:9090"]
30
```

Rys. 2. Plik konfiguracyjny `prometheus.yml`

Źródło: Rysunek autorski.

Jak można zauważyć na Rys.2. plik konfiguracyjny posiada jasno zdefiniowane sekcje, a każda z nich odpowiada za konkretny aspekt działania Prometheusa. Plik w tej formie łatwo można modyfikować i rozszerzać o kolejne sekcje.

3.2. Zabezpieczanie Prometheusa

Podczas konfiguracji Prometheusa, ważnym aspektem są kwestie bezpieczeństwa. Firma, która stworzyła i wspiera to rozwiązanie skupiła się na funkcjonalnościach związanych z monitorowaniem, przez co brakuje w nim wielu funkcjonalności zwiększających bezpieczeństwo[14]. Z tego powodu to od użytkownika zależy, czy do Prometheusa będą miały dostęp wyłącznie osoby, które zostały do tego upoważnione. Skupić się tutaj należy przede wszystkim na dostępie do serwera,

na którym uruchomiony jest Prometheus. Jeśli tego nie zrobimy, to dostęp do bardzo wrażliwych danych mogą uzyskać osoby bez autentykacji poprzez Internet. Od wersji 2.24.0 Prometheus wspiera TLS i Basic authentication poprzez protokół HTTP, jednakże w dokumentacji TLS jest dalej oznaczony jako eksperymentalny[15], a stosowanie Basic authentication bez odpowiedniego szyfrowania, pozwala na przechwycenie loginu i hasła w bardzo prosty sposób.

Inne z ataków, przed którymi się warto zabezpieczyć ze względu na brak wbudowanych zabezpieczeń:

- Denial of Service - ataki, w których poprzez wysyłanie wielu zapytań, program przestaje odpowiadać z powodu braku wolnych zasobów,
- Cross-site request forgery i Cross-site scripting - ataki opierające się na wprowadzeniu w błąd użytkownika, aby nieświadomie wykonał zamiary atakującego,

Ostatnim ważnym aspektem jest stworzenie monitorowanego systemu w taki sposób, aby nie zawierał on w logach informacji tajnych. Jeśli nie zostanie to zrobione, Prometheus może być wykorzystany, aby uzyskać informacje, które mogą być następnie wykorzystane do nieuprawnionego dostępu do monitorowanego systemu.

3.3. Narzędzia wspierające

Główną funkcjonalnością Prometheusa jest zbieranie i przetrzymywanie metryk na temat aplikacji, tworzy również podstawowe wizualizacje danych, jednak, aby stworzyć coś bardziej zaawansowanego wymagane jest skorzystanie z dodatkowego oprogramowania. W tym celu powstała właśnie Grafana[7], która jest popularnym narzędziem zintegrowanym z Prometheusem pozwalającym na tworzenie zaawansowanych wizualizacji. Główną funkcjonalnością tego narzędzia jest tworzenie dashboardów, które mogą zawierać wizualizację w postaci wykresów, tabel czy metryk, a dane do nich mogą być dowolnie filtrowane oraz ograniczane czasowo. Każda z wizualizacji może być dodatkowo personalizowana poprzez zmianę koloru, nazw osi, legendy czy opisów. Jednak nie jest to jedyna funkcjonalność Grafany, ponieważ pozwala ona również na tworzenie wzorców oraz zmiennych, które mogą zostać wykorzystane do tworzenia dynamicznych dashboardów. Warto dodać, że zmienne mogą bazować na zapytaniach, listach czy zakresach czasowych dzięki czemu możliwe jest adaptacja do zmieniających się danych i eksplorowanie ich na wielu poziomach. Grafana posiada również wbudowany moduł odpowiedzialny za alerty, umożliwiający tworzenie alarmów na podstawie metryk oraz dowolnie określonych warunków, a następnie wysłanie powiadomienia na email, Slack czy inny webhooki. Funkcjonalności Grafany mogą być rozszerzane poprzez dodawanie kolejnych pluginów umożliwiających m.in. integrację z innymi serwisami czy nowe transformacje danych, dzięki czemu może być ona dowolnie przystosowywana do wymagań użytkownika.

Rozszerzenie Grafany związane z alarmami można uznać za redundantne, gdyż Prometheus posiada zintegrowany moduł odpowiedzialny za powiadomienia. Modułem tym jest Alertmanager, który za pomocą Prometheus Alertmanager API otrzymuje powiadomienia od serwera Prometheusa, gdy spełnione zostaną odpowiednie warunki alarmu. Alerty bazować mogą na wartościach metryk, progach czy innych warunkach określonych w konfiguracji. Powiadomienie następnie może zostać wysłane na maila lub tak samo jak w przypadku alarmów Grafany na dowolny webhook. Alertmanager spełnia szereg funkcji mających za zadanie ułatwić użytkownikowi. Pozwala na usuwanie duplikatów alarmów oraz ich grupowanie dzięki czemu użytkownik nie otrzymuje zbędnych powiadomień oraz ma możliwość przejrzenia statusu całego systemu w jednym miejscu. W razie awarii lub znanej usterki możliwe jest czasowe wyłączenie powiadomień, dzięki czemu użytkownik nie otrzymuje zbędnych

duplikatów powiadomień w czasie naprawy. Przydatną opcją może być także tworzenie wzorców alertów, zawierających konkretne wartości metryk czy opis. Funkcjonalność ta w dużym stopniu ułatwia tworzenie monitoringu dla nowych aplikacji, dla której tworzenie alertów jest wtedy kwestią wykonania kilku kliknięć myszą. Alertmanager udostępnia oczywiście także interfejs użytkownika, który w znacznym stopniu ułatwia konfigurację i modyfikację alertów oraz pozwala na podejrzenie ich stanu.

Kolejnym rozszerzeniem Prometheusa jest Prometheus Pushgateway[16] pozwalający na wysłanie metryk bezpośrednio do Prometheusa przez np. programy batchowe. Jest to odejście od standardowego modelu, w którym to metryki pobierane są do Prometheusa poprzez odpytywanie specjalnego endpointu. Ma to przede wszystkim zastosowanie w programach batchowych, programach żyjących krótko lub takich które mogą często nie być dostępne w zadanym przedziale czasowym. Dzięki takiemu rozwiązaniu wszystkie aplikacje tego typu mogą wysyłać swoje metryki do jednego miejsca, z którego następnie pobierane są przez Prometheusa i przetwarzane. Oczywiście metryki te muszą zostać wysłane w odpowiednim formacie, do czego przydatne mogą okazać się biblioteki zewnętrzne pozwalające na formatowanie metryk do formatu zgodnego z formatem Prometheusa. Warto zaznaczyć, że Pushgateway nie jest zaprojektowany do trzymania metryk przez długi czas, dlatego w przypadku, gdy mamy potrzebę trzymać metryki przez dłuższy czas powinniśmy skorzystać z innej integracji.

Jedną z takich integracji jest Thanos[17]. Jest to open-sourcowe rozszerzenie do Prometheusa znacznie zwiększające jego możliwości w zakresie przechowywania danych oraz ich przeszukiwania. Największą zaletą Thanosa jest to, że może działać jako sidecar tuż obok Prometheusa, a co za tym idzie przechwytywać dane przez niego zbierane, a następnie wysyłać do różnego rodzaju magazynów danych takich jak Amazon S3, Google Cloud Storage czy innych komponentów zdolnych do przetrzymywania danych. Dzięki temu możliwe jest znaczne zwiększenie retencji metryk i przeglądanie danych historycznych. Thanos dostarcza jedno centralne miejsce, z którego może zostać wykonane zapytanie w celu przeszukania danych z wielu źródeł, ale także z wielu instancji Prometheusa, uzyskując w ten sposób podgląd całego systemu. Rozszerza on także możliwości alarmów dobrze znanych z Prometheusa czy Grafany, dostarczając centralne miejsce, w którym zebrane zostały wszystkie alarmy, a zarazem uzyskując w ten sposób podgląd na stan całego systemu.

Część systemów czy aplikacji może nie generować metryk lub generować je, ale z formatem niezgodnym z formatem Prometheusa. W celu zapewnienia jednolitości i możliwości monitorowania wielu systemów powstały Prometheus Exporters[8], których głównym zadaniem jest zbieranie metryk z wielu systemów, serwisów czy aplikacji i wysyłanie ich w odpowiednim formacie do Prometheusa. Można uznać je za pewnego rodzaju pośredników, zbierających informację na temat zużycia procesora, zużycia pamięci, ruchu internetowego, ale także metryk związanych ze specyfikacją danego systemu, a następnie wystawiających ich na odpowiedni endpoint w formacie zgodnym z Prometheusem. Można wyróżnić kilka rodzajów Exporterów, dla przykładu Node Exporter będzie zbierał dane na temat parametrów systemów Linux, Windows czy innych systemów typu UNIX. Do śledzenia metryk na temat bazy danych możemy użyć m.in. Redis Exporter, PostgreSQL Exporter czy MySQL Exporter, ale możliwe jest także pobieranie metryk z serwera aplikacyjnego Apache za pomocą Apache Exporter czy metryk dotyczących Kafki za pomocą Kafka Exporter.

Prometheus jako narzędzie do monitorowania systemów rozproszonych powinno być przede wszystkim jak najłatwiejsze we wdrożeniu, skalowalne oraz umożliwiające łatwe podłączenie kolejnych modułów i rozszerzeń takich jak Grafana. Funkcja ta na klastrze Kubernetesa zapewniona jest przez narzędzie znane jako Prometheus Operator[18]. Operator upraszcza wdrożenie

Prometheusa na Kubernetes oraz komponentów z nim powiązanych. Dostarcza możliwość definiowania zasobów (CRD - Custom Resource Definitions), operatory automatyzujące wprowadzanie konfiguracji, skalowalność oraz dostępność Prometheusa. Prometheus Operator wykorzystuje Kubernetes API do zarządzania żywotnością i konfiguracją aplikacji, w tym monitoruje oraz utrzymuje żądany stan instancji Prometheusa i zasobów z nim powiązanych. Najważniejszymi zaletami Prometheusa wdrożonego na klastrze Kubernetesa jest skalowalność, dostępność, automatyczne wykrywanie serwisów, ale także samo-napraw, poprzez monitorowanie i wdrażanie poprawek tak, aby aplikacja działała bez błędów oraz aktualizacje nie wymagające zatrzymywania aplikacji, poprzez odpowiednie zarządzanie podami i kontenerami.

Jak zatem można zauważyć Prometheus wspiera szereg modułów rozszerzających jego możliwości. Mogą one zostać wykorzystane do stworzenia monitoringu systemów rozproszonych oraz zebrać informację na temat ich stanu w jednym miejscu.

3.4. Wspierane języki programowania

Prometheus został napisany w języku Go[19], zapewniając stabilny i wydajny system pozwalający na monitorowanie oraz tworzenie zapytań w języku Prometheus Query Language (PromQL). Mimo tego, że Prometheus został napisany w języku Go to wspiera on także inne języki programowania[9], poprzez biblioteki oraz wspomniane wcześniej Prometheus Exporters[8]. Oficjalna biblioteki noszą następujące nazwy: "client_golang" dla języka Go, "client_java" dla języków Javy i Scala, "client_python" dla języka Pythona, "client_ruby" dla języka Ruby oraz "client_rust" dla aplikacji napisanych w języku Rust. Dodatkowo możemy wyróżnić nieoficjalne biblioteki wspierające między innymi języki C, C++, C#, Haskell czy PHP.

Użycie wymienionych bibliotek pozwala na zbieranie danych metrycznych na temat aplikacji, a następnie wystawienie ich na odpowiedni endpoint, skąd Prometheus będzie mógł je pobrać, a następnie przetworzyć. Oczywiście tak jak to zostało opisane wcześniej w celu zbierania metryk z systemów, serwisów czy serwerów aplikacyjnych możemy użyć także Prometheus Exporters, które pełnią rolę pośrednika, zapewniając, że metryki z danej aplikacji czy systemu trafią w odpowiednim formacie na serwer Prometheusa.

Duża ilość bibliotek oraz Prometheus Exporters zapewnia, że każda część systemu rozproszonego niezależnie od tego w jakim języku programowania jest napisana lub jaką architekturą systemu się charakteryzuje może być monitorowana poprzez Prometheusa i narzędzie go wspierające.

3.5. Porównanie Prometheus z konkurencją

Wybierając narzędzie do monitorowania systemu rozproszonego należy zastanowić się przede wszystkim nad jego możliwościami, ale także czasem i poziomem złożoności wdrożenia czy kosztami przygotowania takiego rozwiązania. W tym podrozdziale przyjrzymy się rozwiązaniom konkurencyjnym oraz jakimi zaletami i wadami w stosunku do nich charakteryzuje się Prometheus[20].

Pierwszym rozwiązaniem, któremu warto się przyjrzeć jest Graphite. Jest to najprościej mówiąc baza danych, przechowująca dane oraz tworzącą wykresy na ich podstawie, jednak w odróżnieniu od Prometheusa, Graphite nie zbiera danych sam, a za pomocą zewnętrznych aplikacji. W Graphite dane przechowywane są w postaci próbek czasowych, oddzielonych kropkami. Model danych oferowany przez Prometheusa jest podobny jednak znacznie bogatszy, ponieważ charakteryzuje się on kodowaniem wartości jako pary klucz-wartość, co ułatwia późniejsze filtrowanie

czy grupowanie wartości. Prometheus osiąga także przewagę w przetrzymywaniu danych, gdzie jak zostało to opisane wcześniej jest możliwość integracji Prometheusa z zewnętrznym magazynem danych i zapisywania ich na dłuższy przedział czasu. Graphite zapisuje dane lokalnie na dysku w postaci próbek, które po pewnym przedziale czasu są nadpisywane przez te nowsze. Dodatkowo dane są zapisywane w formacie RRD, który oczekuje, że dane będą przychodzić w regularnych odstępach czasu co może okazać się kłopotliwe w przypadku systemu rozproszonego. Prometheus oferuje znacznie bogatszy model danych, a przy tym jest także łatwiejszy do skonfigurowania i uruchomienia w systemie rozproszonym.

Nagios jest systemem do monitorowania, którego podstawową funkcjonalnością jest alarmowanie na podstawie kodów końcowych skryptów, które nazywane są "checks". Pozwala na wyciszenie pojedynczego alarmu, jednak nie pozwala na grupowanie czy usuwanie duplikatów. Nagios jest bezpośrednio powiązany z danym hostem, każdy z nich może mieć kilka serwisów, a każdy z serwisów może wykonać jeden "check". Rozwiązanie to także nie przechowuje danych poza aktualnym stanem "checks" jednak może to zostać rozszerzone przez dodatkowe pluginy. Nagios, więc będzie odpowiednim rozwiązaniem, jeśli chcemy stworzyć podstawowy monitoring dla małego systemu, w przypadku dużego systemu rozproszonego znacznie prostszym do zaimplementowania rozwiązaniem będzie Prometheus.

Kolejnym rozwiązaniem, któremu warto się przyjrzeć jest Sensu, będące open-sourcowym rozwiązaniem monitoringowym, posiadającym rozwiązanie komercyjne które może zostać rozszerzone o funkcjonalność skalowalności. Rozwiązanie to przede wszystkim skupia się na przetwarzaniu i alarmowaniu na podstawie obserwowania danych w postaci "zdarzeń". Funkcjonalności Sensu mogą zostać rozszerzone poprzez zewnętrzne pluginy o filtrowanie, agregację, transformację czy procesowanie "zdarzeń", w tym także wysyłanie alarmów do innych systemów. W tym kontekście jest to bardzo podobne do Alertmanagera znanego z Prometheusa. Sensu trzyma dane o historycznym i aktualnym statusie zdarzeń we wbudowanej lub zewnętrznej bazie danych. Prometheus i Sensu posiadają kilka wspólnych funkcjonalności, co więcej Sensu posiada wsparcie dla metryk pochodzący z Prometheus Exporters, jednak mają zupełnie odmienne podejście do monitorowania aplikacji. Obie platformy mogą zostać wdrożone w dynamicznym chmurowym środowisku, podłączyć się pod zewnętrzny magazyn danych czy korzystać z rozszerzeń. Sensu sprawdzi się jednak lepiej, kiedy chcemy przetwarzać nie tylko metryki, ale także inne dane oraz kiedy planujemy używać wielu narzędzi do monitorowania. Warto pamiętać, że jest to także wydajniejsza platforma pozwalająca na przetwarzanie różnego rodzaju zdarzeń. Prometheus natomiast sprawdzi się lepiej, gdy zbieramy jedynie metryki, monitorujemy system postawiony w pełni na Kubernetesie oraz jeśli chcemy mieć dostęp do lepszych narzędzi przeszukiwania danych.

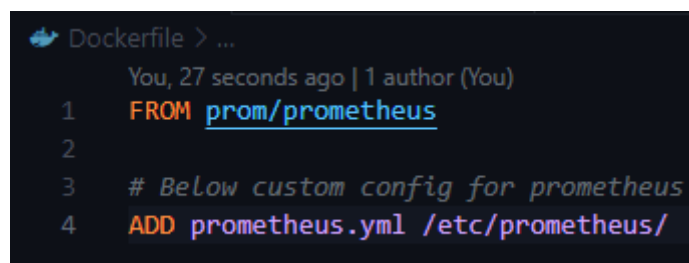
Podsumowując Prometheus na tle konkurencji wypada bardzo dobrze, nie tylko pod względem wydajności, ale także oferowanych funkcjonalności, które mogą dodatkowo być rozszerzane. W swojej niszy Prometheus tak naprawdę może wypatrywać konkurencji jedynie w Sensu, jednak i ta usługa nie oferuje tak bogatego języka zapytań, który pozwoliłby na zapoznanie się z kolekcjonowanymi danymi tak dokładnie jak Prometheus, co przy monitoringu systemów rozproszonych jest dużą zaletą.

4. Przykładowe zastosowanie w projekcie

Link do repozytorium, w którym znajduje się kod i pliki wspomniane w poniższych podrozdziałach: <https://github.com/NorbertPiecka/ORWKiG>.

4.1. Uruchomienie systemu Prometheus z wykorzystaniem kontenera Docker

Jednym z prostszych sposobów uruchomienia Prometheusa jest wykorzystanie dockerowych kontenerów. W tym celu napisany został plik Dockerfile przedstawiony na Rys.3.



```
Dockerfile > ...
You, 27 seconds ago | 1 author (You)
1 FROM prom/prometheus
2
3 # Below custom config for prometheus
4 ADD prometheus.yml /etc/prometheus/
```

Rys.3. Zrzut ekranu pliku Dockerfile Prometheusa.

Źródło: Rysunek autorski.

Jak można zauważyć na powyższym zrzucie ekranu struktura jest bardzo prosta. W pierwszej kolejności pobierany jest oficjalny obraz prometheusa, a następnie w linii 4 dodawany jest w odpowiednim miejscu zmodyfikowany plik konfiguracyjny prometheus.yml. Plik ten zawiera informacje na temat adresów endpointów, z których Prometheus ma pobierać dane i został przedstawiony na Rys.4.



```
! prometheus.yml
...
1 global:
2   scrape_interval: 15s
3   evaluation_interval: 15s
4 alerting:
5   alertmanagers:
6     - static_configs:
7       - targets:
8         # - alertmanager:9093
9
10 rule_files:
11   #rule_1
12
13 scrape_configs:
14   - job_name: "java-app"
15     static_configs:
16       - targets: ["192.168.1.12:8002"]
17
18   - job_name: "python-app"
19     static_configs:
20       - targets: ["192.168.1.12:8000"]
21
22   - job_name: "prometheus"
23     static_configs:
24       - targets: ["localhost:9090"]
```

Rys.4. Zrzut ekranu pliku prometheus.yml.

Źródło: Rysunek autorski.

Plik składa się z kilku sekcji, pierwsza z nich dotyczy globalnych ustawień prometheusa, możemy tam m.in. ustawić odstęp czasu, w którym odpytywane będą endpointy. Z punktu widzenia projektu najważniejsza wydaje się sekcja "scrape_configs", w której zdefiniowane są endpointy. Pierwsze dwa "java-app" i "python-app" odnoszą się do aplikacji napisanych odpowiednio w języku java i python, trzeci endpoint wskazuje bezpośrednio na promethuesa, ponieważ domyślnie zbiera on informacje na temat swoich metryk. Przy budowaniu obrazu wykorzystywany jest również plik docker-compose.yml przedstawiony na Rys.5.



```
docker-compose.yml
...
1  version: "3.9"
2  services:
3    prom:
4      build: .
5      container_name: prometheus
6      ports:
7        - 9090:9090
8      networks:
9        - dev
10 networks:
11   dev:
12     driver: bridge
```

Rys.5. Zrzut ekranu pliku docker-compose.yml.

Źródło: Rysunek autorski.

Plik wykorzystuje przedstawiony wcześniej plik Dockerfile, nadaje nazwę kontenerowi oraz konfiguruje porty, na których będzie działać aplikacja. W celu zbudowania obrazu Prometheusa i uruchomienia go w postaci kontenera wykorzystane zostało polecenie "docker compose up". Po wystartowaniu kontenera jest możliwe dostanie się do panelu nawigacyjnego Prometheusa poprzez wejście na adres "localhost:9090", a następnie do jego kolejnych funkcjonalności poprzez wybór odpowiedniej zakładki z górnego paska.

4.2. Połączenie do Prometheus z wykorzystaniem języków programowania Java i Python

Do pokazania w jaki sposób można używać Prometheusa, stworzyliśmy dwie proste aplikacje odpowiednio w językach Java i Python, które pokazują w jaki sposób można pobrać wykorzystanie procesora, pamięci lub stworzyć licznik.

4.2.1. Java

Pierwszym krokiem jest zainstalowanie i zaimportowanie oficjalnych bibliotek w każdym z języków. Wykorzystaliśmy narzędzie Maven, w którym dodaliśmy zależność w następujący sposób:

```
<dependency>
  <groupId>io.prometheus</groupId>
  <artifactId>simpleclient</artifactId>
  <version>0.16.0</version>
</dependency>
<dependency>
  <groupId>io.prometheus</groupId>
  <artifactId>simpleclient_common</artifactId>
  <version>0.16.0</version>
</dependency>
<dependency>
  <groupId>io.prometheus</groupId>
  <artifactId>simpleclient_hotspot</artifactId>
  <version>0.16.0</version>
</dependency>
<dependency>
  <groupId>io.prometheus</groupId>
  <artifactId>simpleclient_httpserver</artifactId>
  <version>0.16.0</version>
</dependency>
```

Następnie w pliku z kodem zaimportowaliśmy miernik, licznik oraz serwer eksportera:

```
import io.prometheus.client.Counter;
import io.prometheus.client.Gauge;
import io.prometheus.client.exporter.HTTPServer;
```

W kolejnym kroku zdefiniowaliśmy jako argumenty statyczne klasy odpowiednie mierniki i licznik:

- licznik requestCounter

```
private static final Counter requestsCounter = Counter.build()
    .name("myapp_requests_total")
    .help("Total number of requests processed")
    .register();
```
- miernik cpuUsageMetric do zapisu użycia procesora

```
private static final Gauge cpuUsageMetric = Gauge.build()
    .name("cpu_usage")
    .help("CPU usage percentage")
    .register();
```

- mierniki heapMemoryUsageMaxMetric oraz heapMemoryUsageUsedMetric do zapisu maksymalnej i obecnie używanej pamięci stosu
- ```
private static final Gauge heapMemoryUsageMaxMetric = Gauge.build()
 .name("heap_memory_usage_max")
 .help("Heap memory usage max in bytes")
 .register();
```

```
private static final Gauge heapMemoryUsageUsedMetric =
Gauge.build()
 .name("heap_memory_usage_used")
 .help("Heap memory usage used in bytes")
 .register();
```

Później stworzyliśmy serwer, który na porcie 8002 będzie udostępniać nasze metryki w następujący sposób:

```
HTTPServer server = new HTTPServer(8002);
```

Należy także pamiętać, aby ten serwer został zamknięty, gdy aplikacja kończy pracę. Można to zrobić za pomocą:

```
server.close();
```

Aby metryki miały poprawne wartości należy je uzupełnić. Dla licznika requestCounter wystarczy użyć metody inc(), która spowoduje zwiększenie tego licznika o 1:

```
requestsCounter.inc();
```

W przypadku mierników natomiast wykorzystuje się metodę set(), którą wykorzystaliśmy do wpisania odpowiednich metryk na temat procesora i pamięci, które uzyskaliśmy z wykorzystaniem wbudowanej biblioteki java.lang.management:

```
cpuUsageMetric.set(cpuUsage);
```

```
...
```

```
heapMemoryUsageUsedMetric.set(usedHeapMemory);
```

```
heapMemoryUsageMaxMetric.set(maxHeapMemory);
```

Po uruchomieniu całego kodu i wejściu w przeglądarce pod adres localhost:8002

widoczne są następujące metryki:

```
HELP cpu_usage CPU usage percentage
```

```
TYPE cpu_usage gauge
```

```
cpu_usage 0.18350782686425868
```

```
HELP myapp_requests_total Total number of requests processed
```

```
TYPE myapp_requests_total counter
```

```
myapp_requests_total 21.0
```

```
HELP heap_memory_usage_used Heap memory usage used in bytes
```

```
TYPE heap_memory_usage_used gauge
```

```
heap_memory_usage_used 1.2582912E7
```

```
HELP heap_memory_usage_max Heap memory usage max in bytes
```

```
TYPE heap_memory_usage_max gauge
```

```
heap_memory_usage_max 4.234149888E9
```

```
HELP myapp_requests_created Total number of requests processed
```

```
TYPE myapp_requests_created gauge
```

```
myapp_requests_created 1.685647951301E9
```

### 4.2.2. Python

Dla języka Python do zainstalowania oficjalnej biblioteki Prometheusa wykorzystaliśmy wbudowany moduł pip w następujący sposób:

```
python -m pip install prometheus-client
```

Następnie z zainstalowanej biblioteki zaimportowaliśmy w pliku z kodem funkcję pozwalającą uruchomić serwer metryk oraz klasy miernika i licznika:

```
from prometheus_client import start_http_server, Gauge, Counter
```

Później stworzyliśmy miernik wykorzystania procesora, miernik wykorzystania pamięci oraz licznik sekund od rozpoczęcia działania programu:

```
cpu_usage_metric = Gauge('cpu_usage', 'CPU usage percentage')
memory_usage_metric = Gauge('memory_usage', 'Memory usage percentage')
seconds_since_start_counter = Counter('seconds_since_start', 'Number of seconds since start')
```

W kolejnym kroku uruchomiliśmy serwer metryk na porcie 8000 za pomocą:

```
start_http_server(8000)
```

Zwiększenie licznika i przypisanie wartości do miernika wyglądało identycznie jak w języku Java - wykorzystywane są do tego metody odpowiednio inc() oraz set():

```
seconds_since_start_counter.inc()
...
cpu_usage_metric.set(cpu_usage)
memory_usage_metric.set(memory_usage)
```

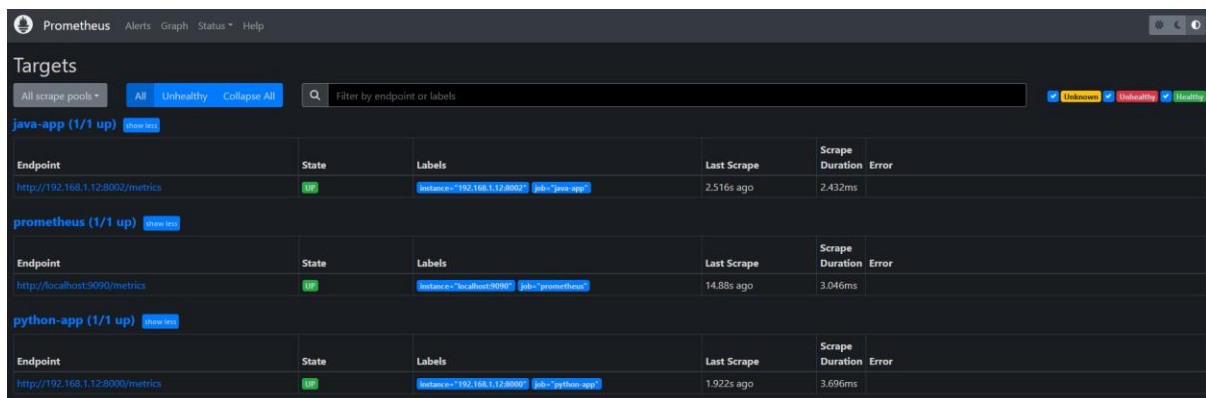
Do pobrania użycia procesora i pamięci wykorzystaliśmy moduł psutil. Po uruchomieniu programu i udaniu się pod adres localhost:8000 uzyskaliśmy dostęp do metryk. Dla języka Python, w przeciwieństwie do Javy, domyślnie dodawane są dodatkowe metryki związane z odświeżaniem pamięci:

```
HELP python_gc_objects_collected_total Objects collected during gc
TYPE python_gc_objects_collected_total counter
python_gc_objects_collected_total{generation="0"} 70.0
python_gc_objects_collected_total{generation="1"} 322.0
python_gc_objects_collected_total{generation="2"} 0.0
HELP python_gc_objects_uncollectable_total Uncollectable object
found during GC
TYPE python_gc_objects_uncollectable_total counter
python_gc_objects_uncollectable_total{generation="0"} 0.0
python_gc_objects_uncollectable_total{generation="1"} 0.0
python_gc_objects_uncollectable_total{generation="2"} 0.0
HELP python_gc_collections_total Number of times this generation was
collected
TYPE python_gc_collections_total counter
python_gc_collections_total{generation="0"} 43.0
python_gc_collections_total{generation="1"} 3.0
python_gc_collections_total{generation="2"} 0.0
HELP python_info Python platform information
TYPE python_info gauge
```

```
python_info{implementation="CPython",major="3",minor="10",patchlevel="4",version="3.10.4"} 1.0
HELP cpu_usage CPU usage percentage
TYPE cpu_usage gauge
cpu_usage 31.7
HELP memory_usage Memory usage percentage
TYPE memory_usage gauge
memory_usage 6.276202496e+09
HELP seconds_since_start_total Number of seconds since start
TYPE seconds_since_start_total counter
seconds_since_start_total 2260.0
HELP seconds_since_start_created Number of seconds since start
TYPE seconds_since_start_created gauge
seconds_since_start_created 1.6856455768659105e+09
```

### 4.3. Przegląd metryk w Prometheusie

W celu sprawdzenia poprawności wykonanego rozwiązania uruchomiony został kontener Prometheusa oraz obie aplikacje. Po przejściu na adres localhost:9090/targets możemy zauważyć, że wskazana endpointy są monitorowane, zrzut ekranu przedstawiający wskazaną stronę znajduje się na Rys.6.

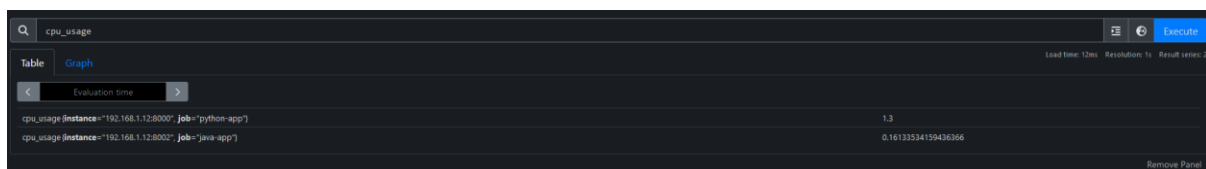


| Endpoint                         | State | Labels                                        | Last Scrape | Scrape Duration | Error |
|----------------------------------|-------|-----------------------------------------------|-------------|-----------------|-------|
| <b>java-app (1/1 up)</b>         |       |                                               |             |                 |       |
| http://192.168.1.12:8000/metrics | up    | instance="192.168.1.12:8000" job="java-app"   | 2.516s ago  | 2.432ms         |       |
| <b>prometheus (1/1 up)</b>       |       |                                               |             |                 |       |
| http://localhost:9090/metrics    | up    | instance="localhost:9090" job="prometheus"    | 14.88s ago  | 3.046ms         |       |
| <b>python-app (1/1 up)</b>       |       |                                               |             |                 |       |
| http://192.168.1.12:8000/metrics | up    | instance="192.168.1.12:8000" job="python-app" | 1.922s ago  | 3.696ms         |       |

**Rys.6.** Zrzut ekranu przedstawiający stronę pod adresem localhost:9090/targets

**Źródło:** Rysunek autorski.

W celu sprawdzenia poprawności przesyłanych metryk zostały wykonane zapytanie o zużycie procesora dla obu aplikacji. Na Rys.7. możemy zobaczyć wynik wyszukiwania w postaci tabeli.

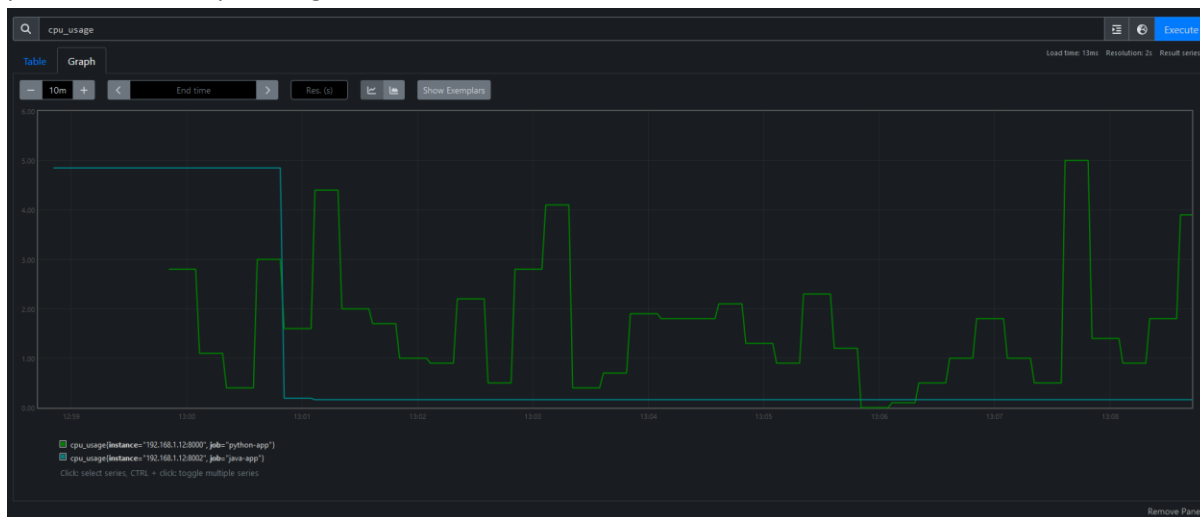


| Series                                                    | Value               |
|-----------------------------------------------------------|---------------------|
| cpu_usage{instance="192.168.1.12:8000", job="python-app"} | 1.3                 |
| cpu_usage{instance="192.168.1.12:8000", job="java-app"}   | 0.16133334159436366 |

**Rys.7.** Zrzut ekranu przedstawiający tabelę po wykonaniu zapytania w języku PromQL.

**Źródło:** Rysunek autorski.

Zrzut ekranu na Rys.8. przedstawia natomiast to samo zapytanie, ale jego wyniki zostały przedstawione w postaci grafu:



**Rys.8.** Zrzut ekranu przedstawiający graf po wykonaniu zapytania w języku PromQL.

**Źródło:** Rysunek autorski.

Oczywiście możliwe jest także filtrowanie danych z użyciem języka PromQL, zapytanie takie zostało wykonane na Rys.9. i przedstawia dane dotyczące zużycia procesora dla aplikacji napisanej w języku Python.



**Rys.9.** Zrzut ekranu przedstawiający graf zużycia pamięci dla aplikacji napisanej w Pythonie.

**Źródło:** Rysunek autorski.

Jak można zauważyć Prometheus czyta poprawnie przesyłane przez aplikacje metryki, a informację w nich zawarte mogą być przeglądane przy użyciu narzędzi wbudowanych w Prometheusa, ale także być w przyszłości użyte do stworzenia alarmów informujących o stanie systemu.

## 5. Podsumowanie i wnioski

Monitorowanie systemów rozproszonych może wydawać się skomplikowane, lecz jest jednocześnie bardzo potrzebne. Dzięki monitorowaniu zachowania systemu mamy pewność, że jeśli dzieje się cokolwiek złego, to możemy zareagować niemal natychmiastowo. Dlatego tak ważne jest dobranie odpowiedniego narzędzia.

Prometheus to narzędzie pozwalające zbierać metryki na temat wielu systemów, jednocześnie będąc przy tym łatwym w konfiguracji. Wspiera on najpopularniejsze języki programowania poprzez oficjalne biblioteki, ale istnieje wiele nieoficjalnych bibliotek dla reszty języków programowania. Uruchomienie Prometheusa przy wykorzystaniu Dockera jest bardzo proste i wymaga znajomości podstawowych komend oraz stworzenia pliku konfiguracyjnego. Należy jednak pamiętać, aby zadbać o zabezpieczenie Prometheusa przed atakami z zewnątrz oraz nieautoryzowanym dostępem, ponieważ nie posiada on wielu funkcjonalności związanych z bezpieczeństwem.

Przedstawiony w rozdziale czwartym projekt, potwierdza, że konfiguracja Prometheusa jest bardzo prosta, tak samo jako konfiguracja aplikacji, aby wysyłały metryki w odpowiednim formacie. Zbieranie informacji na temat wskazanej aplikacji wymaga jedynie dodanie sekcji w pliku konfiguracyjnym `prometheus.yml`, a aplikowanie bibliotek odpowiedzialnych za zbieranie oraz wystawianie metryk na odpowiedni endpoint jest kwestią kilku dodatkowych linii kodu. Dodatkowym atutem Prometheusa jest także czysty interfejs graficzny ułatwiający przeglądanie zebranych danych oraz prosty w użyciu język zapytań PromQL.

Prometheus sprawdzi się świetnie jako narzędzie do monitorowania systemów rozproszonych, możliwość uruchomienia go jako kontener sprawia, że staje się on narzędziem niezależnym od systemu operacyjnego. Mnogość rozszerzeń oraz wsparcie dla różnych języków programowania wyróżnia go na tle konkurencji nie tylko pod względem funkcjonalności, ale także możliwości adaptacji do konkretnych potrzeb monitorowanego systemu. Warto wspomnieć, także że w czasach wzrostu znaczenia rozwiązań chmurowych, a jednocześnie rozwiązań opartych o technologię konteneryzacji oraz mikroservisów Prometheus ma szansę jeszcze bardziej się rozwinąć i stać się narzędziem wykorzystywanym powszechnie do monitorowania systemów rozproszonych.

## 6. Bibliografia

- [1] Zettler, K. (31.05.2023). *Czym jest system rozproszony?* Pobrano z: <https://www.atlassian.com/pl/microservices/microservices-architecture/distributed-architecture#:~:text=System%20rozproszony%20to%20zbi%C3%B3r%20program%C3%B3w,gard%C5%82a%20lub%20centralne%20punkty%20awarii>
- [2] Dokumentacja, Prometheus. (31.05.2023). *Getting Started*. Pobrano z: [https://prometheus.io/docs/prometheus/latest/getting\\_started/](https://prometheus.io/docs/prometheus/latest/getting_started/)
- [3] Dokumentacja, Prometheus. (31.05.2023). *Storage*. Pobrano z: <https://prometheus.io/docs/prometheus/latest/storage/>
- [4] Dokumentacja, Prometheus. (31.05.2023). *Querying Prometheus*. Pobrano z: <https://prometheus.io/docs/prometheus/latest/querying/basics/>
- [5] Dokumentacja, Prometheus. (31.05.2023). *Alerting Rules*. Pobrano z: [https://prometheus.io/docs/prometheus/latest/configuration/alerting\\_rules/](https://prometheus.io/docs/prometheus/latest/configuration/alerting_rules/)
- [6] Dokumentacja, Prometheus. (31.05.2023). *Configuration*. Pobrano z: <https://prometheus.io/docs/alerting/latest/configuration/>
- [7] Dokumentacja, Prometheus. (31.05.2023). *Grafana Support For Prometheus*. Pobrano z: <https://prometheus.io/docs/visualization/grafana/>
- [8] Dokumentacja, Prometheus. (31.05.2023). *Exporters And Integrations*. Pobrano z: <https://prometheus.io/docs/instrumenting/exporters/>
- [9] Dokumentacja, Prometheus. (31.05.2023). *Client Libraries*. Pobrano z: <https://prometheus.io/docs/instrumenting/clientlibs/>
- [10] Dokumentacja, Prometheus. (31.05.2023). *Federation*. Pobrano z: <https://prometheus.io/docs/prometheus/latest/federation/>
- [11] Dokumentacja, Prometheus. (31.05.2023). *Overview*. Pobrano z: <https://prometheus.io/docs/introduction/overview/>
- [12] Dokumentacja, Prometheus. (31.05.2023). *HTTP API*. Pobrano z: <https://prometheus.io/docs/prometheus/latest/querying/api/>
- [13] Dokumentacja, Prometheus. (31.05.2023). *Exposition Formats*. Pobrano z: [https://prometheus.io/docs/instrumenting/exposition\\_formats/](https://prometheus.io/docs/instrumenting/exposition_formats/)
- [14] Arghire, I (31.05.2023). *Many Prometheus Endpoints Expose Sensitive Data*. Pobrano z: <https://www.securityweek.com/many-prometheus-endpoints-expose-sensitive-data/>
- [15] Dokumentacja, Prometheus. (31.05.2023). *Security*. Pobrano z: <https://prometheus.io/docs/operating/security/>
- [16] Dokumentacja, Prometheus. (31.05.2023). *When To Use The Pushgateway*. Pobrano z: <https://prometheus.io/docs/practices/pushing/>
- [17] Dokumentacja, Thanos (31.05.2023), *Getting Started*. Pobrano z: <https://thanos.io/tip/thanos/getting-started.md/>
- [18] Repozytorium, Prometheus Operator (31.05.2023), <https://github.com>. Pobrano z: <https://github.com/prometheus-operator/prometheus-operator>

- [19] Dokumentacja, Prometheus. (31.05.2023). *Frequently Asked Questions*. Pobrano z: <https://prometheus.io/docs/introduction/faq/>
- [20] Dokumentacja, Prometheus. (31.05.2023). *Comparison To Alternatives*. Pobrano z: <https://prometheus.io/docs/introduction/comparison/>