Wdrożenie Jitsi na platformie Kubernetes, zbadanie wydajności oraz próba optymalizacji

Dokumentacja projektu

Jakub Jędrzejczyk Sebastian Kwaśniak Anna Berkowska

2025-12-11

Spis treści

Wstęp teoretyczny

Kubernetes

Kubernetes to otwartoźródłowy system orkiestracji kontenerów, opracowany pierwotnie przez Google, a obecnie rozwijany przez Cloud Native Computing Foundation (CNCF). Umożliwia on automatyzację wdrażania, skalowania i zarządzania aplikacjami kontenerowymi. Kubernetes abstrahuje fizyczną infrastrukturę, udostępniając logiczny zestaw zasobów.

Jedną z kluczowych cech Kubernetes jest jego dynamiczny charakter - zasoby i kontenery mogą być tworzone, usuwane i przenoszone pomiędzy węzłami klastra w sposób automatyczny. Ta elastyczność zwiększa odporność systemu, ale jednocześnie utrudnia monitorowanie jego stanu i wydajności przez typowe rozwiązania monitoringu aplikacji.

W Kubernetesie wszystkie obiekty można podzielić na dwa: klastrowe oraz ograniczone do przestrzeni nazw. Te drugie są wyizolowane logicznie od siebie, dzięki czemu można nazywać tak samo, oraz tworzyć ograniczenia pomiędzy przestrzeniami nazw tak, aby aplikacje nie miały do siebie nawzajem dostępu, co uczyni klaster bezpieczniejszym. Niektóre z obiektów mogą być tylko klastrowe, a niektore tylko w konkretnej przestrzeni nazw.

Każdy element Kubernetes posiada metadane, oraz właściwe parametry (przeważnie, lecz nie zawsze zdefiniowane jako `spec:` w pliku YAML). Metadane zawierają m.in nazwę obiektu oraz przestrzeń nazw w której się znajduje.

Sieć w Kubernetes

W Kubernetes sieć jest zaprojektowana jako płaska struktura, w której każdy Pod otrzymuje swój własny, unikalny adres IP. Dzięki temu Pody mogą komunikować się ze sobą bezpośrednio, bez potrzeby używania NAT (tłumaczenia adresów) wewnątrz klastra.

Kluczowe zasady działania:

- Wymagany Plugin (CNI): Kubernetes sam w sobie nie zapewnia warstwy sieciowej. Musisz wybrać i zainstalować wtyczkę CNI (Container Network Interface), taką jak Calico, Flannel czy Cilium. To ten plugin odpowiada za przydzielanie adresów IP i routing pakietów między Podami.
- Komunikacja Pod-Pod: Niezależnie od tego, na którym węźle (Node) znajdują się Pody, widzą się one tak, jakby były w tej samej sieci lokalnej.
- Ruch na zewnątrz (Bypassing Master): Gdy aplikacja na Worker Node chce połączyć się z Internetem lub zewnętrzną usługą, ruch wychodzi bezpośrednio z tego Workera.
- Ważne: Pakiety danych pomijają Master Node (Control Plane). Master służy jedynie do zarządzania klastrem (API, scheduler), ale nie pośredniczy w przesyłaniu danych użytkowych, nawet jeśli są na nim hostowane komponenty systemowe.

Jitsi

Jitsi to platforma wideokonferencyjna open-source oparta na standardzie WebRTC. Jej działanie opiera się na architekturze SFU (Selective Forwarding Unit). Oznacza to, że serwer nie "miesza" obrazu wszystkich uczestników w jeden strumień (co wymagałoby ogromnej mocy obliczeniowej), lecz przekazuje

(forwarduje) odpowiednie pakiety wideo od jednego użytkownika do pozostałych. Jitsi posiada wiele komponentów, ale omówimy tylko dwa najważniejsze z nich:

Kluczowe komponenty:

- JVB (Jitsi Videobridge) - JVB odbiera strumienie audio/wideo od użytkowników i przesyła je dalej.
  - Simulcast: JVB dba o jakość - przy słabym łączu JVB wysyła strumień o niższej jakości, nie obciążając przy tym nadawcy.
- Jicofo (Jitsi Conference Focus) - Zarządza sesjami konferencyjnymi.
  - Działanie: Nie dotyka mediów (obrazu/dźwięku). Zajmuje się logiką: pilnuje, kto jest w pokoju, kto ma prawo głosu, i przydziela uczestników do konkretnego mostka (JVB).
  - Sygnalizacja: Komunikuje się z użytkownikami (zazwyczaj przez serwer XMPP, np. Prosody) i instruuje JVB, co ma robić.

Jitsi Octo (Relay)

W standardowym Jitsi wszyscy uczestnicy spotkania muszą być podłączeni do jednego mostka (JVB).

Octo pozwala połączyć wiele instancji JVB w jeden logiczny klaster. W trybie Octo, JVB zachowuje się jednocześnie jak serwer (dla użytkowników) i jak klient (dla innych mostków JVB), przekazując pakiety między JVB. To nadal Jicofo decyduje gdzie użytkownik się połączy.

TODO: sebsk: opisac to co sie dowiedzielismy o jitsi octo TODO: doppie: opisac to co sie dowiedzielismy o jitsi octo

Wybór narzędzi

Do postawienia klastra użyjemy programu Ansible.

Narzędzia z których korzystamy do zarządzania klastrem:

- Helm - do prostego wdrażania dużych aplikacji,
- kubectl - domyślny program do interkacji z klastrami Kubernetes,
- k9s - do testowania oraz szybszego sprawdzania informacji na klastrze.

Programy które wdrożymy na klastrze w kontenerach:

- Jitsi - nasz zestaw programów do wideokonferencji,
- Alloy - do zbierania metryk z Jitsi (dokładniej komponentu Videobridge),
- Mimir - baza danych do zbierania metryk,
- Grafana - do wizualizacji wstępnej (nie do tego dokumentu, do live demo) oraz prostszego wyeksportowania danych.

Jako, że nie posiadamy publicznego adresu IP, wybraliśmy także następujące narzędzia do udostępnienia Jitsi:

- Tailscale - TODO: doppie: opisać (krótko),
- Pangolin: TODO: doppie: opisać (krótko). TODO: doppie: jak czegos zapomnialem to dodaj pls

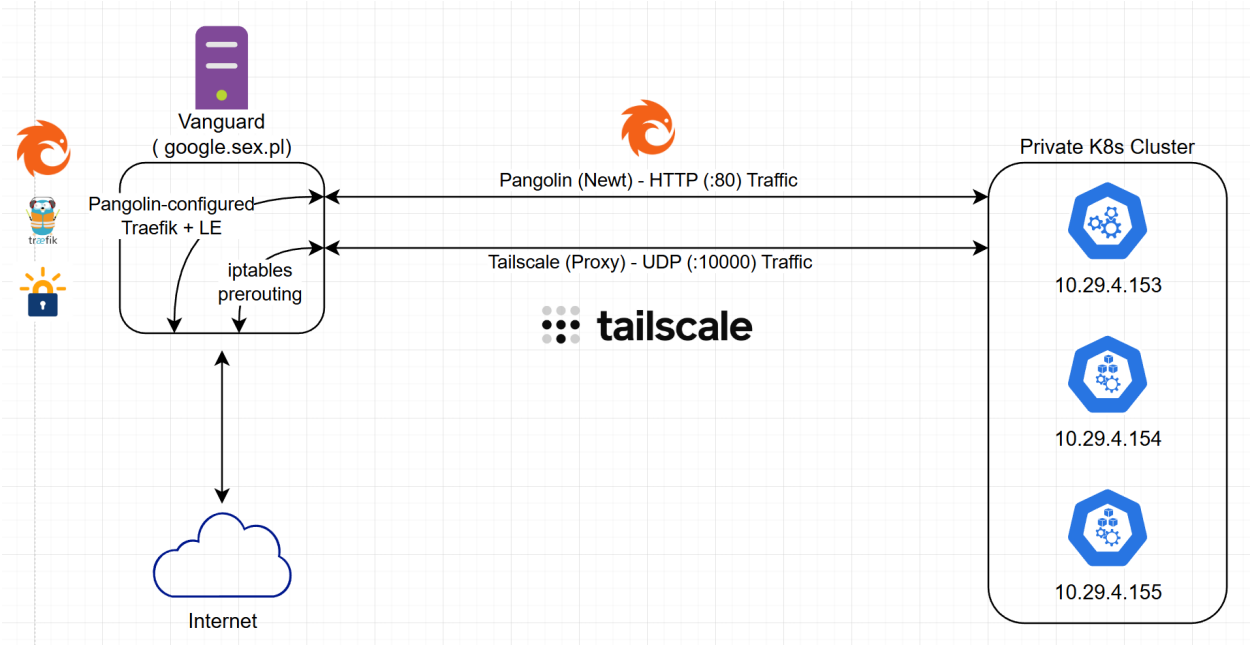
Do prostszego dołączenia wielu użytkowników skorzystamy z platformy `node.js` oraz dostępnej na niej biblioteki `puppeteer`.

Scenariusz - projekt

Sieć

TODO: doppie: opisać tutaj sieć (czemu i co), bez mowienia jak ją postawić (to bedzie dalej)

Sieć została zwizualizowana na rysunku 1.

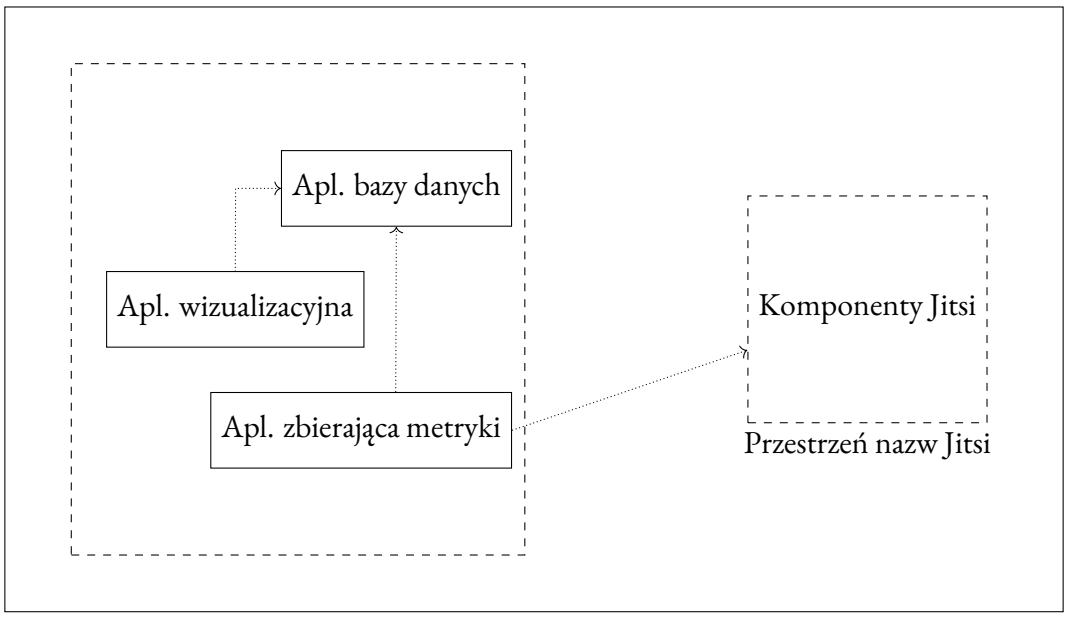


Rysunek 1: Projekt sieci do udostępnienia Jitsi publicznie

Zebranie danych

Mając zdefiniowane Jitsi, przejdźmy do monitoringu. Chcemy, aby zbierał on metryki, po czym wysyłał je do bazy danych. Dodatkowo, chcemy mieć możliwość prostego przeglądania tych danych i tworzenia paneli wizualizacyjnych.

Wszystko to wizualizuje rysunek 2.



Rysunek 2: Scenariusz monitoringu

Replikacja środowiska

Najprostszym sposobem jest użycie jednego z narzędzi: K3S, Minikube, Kind. Programy te upraszczają zestawienie środowiska Kubernetes, upraszczając większość niezbędnych konfiguracji. Można wtedy przejść od razu do punktu z wdrożeniem Jitsi.

> Uwaga! Korzystając z jednego z tych narzędzi, jest bardzo wysokie prawdopodobieństwo, że Octo (Relay) nie będzie działał!

Oprócz samego Kubernetes i domyślnego narzędzia `kubectl`, przyda się również `helm`, które jest szeroko stosowanym menadżerem paczek dla Kubernetes.

Maszyny

W naszym przypadku skorzystaliśmy z trzech maszyn wirtualnych:

- maszyna o adresie 10.29.4.153 (control plane): 2 rdzenie, 2GB RAMu;
- maszyny o adresach 10.29.4.154, 10.29.4.155 (worker): 4 rdzenie, 4GB RAMu.

Kubernetes

Nasze środowisko zostało wdrożone za pomocą przygotowanych przez nas skryptów Ansible, które dostępne są w repozytorium: https://github.com/SebSK3/uiam-prepare-k8s

Najważniejszym elementem jest przygotowanie maszyn (node), których wartości można znaleźć w folderu `host_vars/`. Użytkownik, powinien dostosować następujące wartości:

- ansible_host - lokalny adres IP maszyny,
- ansible_user - użytkownik do którego będzie się podłączać skrypt ansible,
- ansible_ssh_pass - hasło do maszyny,
- ansible_ssh_port - port na którym nasłuchuje SSH na docelowej maszynie.

Jitsi

TODO: doppie:

Udostępnienie Jitsi publicznie

TODO: doppie: można wrzucić screeny, przykład masz w rozdziale Scenariusz - projekt->sieć, a wszystkie scr są w folderze ./res

Monitoring

Do zweryfikowania prędkości wdrożymy prostą instancję Grafana (Mimir - bazy danych szeregów czasowych, Alloy - do przesyłania metryk, Grafana - do wizualizacji i przerobienia danych).

Baza danych

Do przechowywania metryk został wybrany program Mimir w trybie mikroserwisów.

Został on skonfigurowany w następujący sposób:

TODO: folissa:

Listing 1: Plik values.yaml programu Mimir

Zostało to tak skonfigurowane, ponieważ w przypadku testowania niepotrzebne jest nam długotrwałe przechowywanie metryk. Wiązałoby to się z dodatkowym obciążeniem klastra, a dodatkowo z dużą warstwą którą musielibyśmy dodatkowo wdrożyć. Metryki te są przechowywane w pamięci operacyjnej i znikają, gdy Pody się zrestartują.

Wdrożenie go następuje za pomocą komend:

```
$ komenda1 TODO: folissa:
$ komenda2
```

Komendy robią następująco: TODO: folissa:

1. Dodanie coś tam coś
2. Coś tam coś tam

Program do zbierania i wysyłania metryk

Program do wysyłania i zbierania metryk w naszym wypadku to Alloy.

Na początku skonfigurujemy nasz program, poprzez obiekt Kubernetes ConfigMap, który trzyma dane na zasadzie klucz: wartość.

```
---
apiVersion: "v1"
kind: "ConfigMap"
metadata:
  name: "alloy-config-map"
  namespace: "alloy"
data:
  config: |-
      remote.kubernetes.secret "credentials" {
        name = "monitoring-secret"
        namespace = "alloy"
      }
      prometheus.remote_write "mimir" {
        endpoint {
          url = "http://mimir-gateway.monitoring.svc:80/api/v1/push"
          basic_auth {
            username = "admin"
            password = remote.kubernetes.secret.credentials.data["password"]
          }
        }
      }
      prometheus.operator.servicemonitors "jitsi" {
        forward_to = [prometheus.remote_write.mimir.receiver]
        namespaces = ["default"]
      }
```

Listing 2: Obiekt ConfigMap dla programu Alloy

Idąc od góry:

1. Sekcja remote.kubernetes.secret definiuje, że dane autoryzujące, z których później będzie korzystać, będą w obiekcie typu Secret, o nazwie monitoring-secret w przestrzeni nazw Alloy.
2. Sekcja prometheus.remote_write definiuje gdzie mają być przesyłane dane. Definiują endpoint, czyli punkt końcowy, gdzie podany jest URL, oraz dane do autoryzacji. Jak można zauważyć, hasło pobierane jest z wcześniej zdefiniowanego sekretu, o kluczu "password".
3. Sekcja prometheus.operator.servicemonitors określa typową statyczną aplikację (w naszym wypadku Jitsi). Wskazuje przestrzeń nazw, gdzie znajduje się ServiceMonitor z którego ma korzystać do monitorowania aplikacji.

Aby wdrożyć ConfigMap wystarczy zapisać go w pliku i wdrożyć za pomocą kubectl apply -f <nazwa_pliku>.

Do wdrożenia samego programu skorzystamy z prostych wartości, które zapiszemy w pliku values.yaml (dla wygody w innym folderze niż Mimir):

```
---
alloy:
  configMap:
    create: false
    name: "alloy-config-map"
    key: "config"
serviceMonitor:
  enabled: false
```

Listing 3: Wartości values.yaml dla konfiguracji programu Alloy

To, co robi ten fragment wartości, to:

1. Wyłącza tworzenie domyślnej ConfigMap (która zawiera konfigurację programu).
2. Wskazuje na stworzoną przez nas ConfigMap.
3. Wskazuje klucz w którym znajduje się faktyczna wartość w której istnieje konfiguracja.
4. Wyłącza domyślny serviceMonitor, który jest tworzony aby Alloy mógł monitorować sam siebie (na potrzeby testów, aby nie zaśmiecać niepotrzebnymi danymi naszego eksperymentowania).

Wdrożenie jest analogiczne do programu Mimir:

```
$ helm install alloy grafana/alloy --namespace alloy --values values.yaml
```

Aplikacja prezentowania danych

Do monitorowania danych wykorzystywany jest program Grafana. Skonfigurowana została ona w następujący sposób

TODO: folissa:

Listing 4: Plik values.yaml programu Grafana

Zostało to tak skonfigurowane, ponieważ coś tam coś. Fragment cośtamcoś odpowiada cośtamcoś

Wdrożenie go następuje za pomocą komend:

```
$ komenda1 TODO: folissa:
$ komenda2
```

Komendy robią następująco: TODO: folissa:

1. Dodanie coś tam coś
2. Coś tam coś tam

Aby połączyć się z programem Grafana należy przekierować port z Kubernetes na swoją maszynę:

```
$ kubectl port-forward service/grafana 3000:3000 -n <przestrzen_nazw_grafany> TODO: folissa: poprawic
```

> Uwaga! Wymaga to, abyśmy mieli dostęp do klastra z naszej maszyny.

Dodatkowo, został przygotowany panel, który wizualizuje dynamiczną aplikację. Jest dołączony do plików, które zostały wysłane na enauczaniu (jest za długi aby umieścić go tutaj). Ma dwie wizualizacje: pierwszą,

która wizualizuję sumę zapytań do wszystkich wdrożonych aplikacji, oraz drugą, sumę każdej instancji z aplikacji z osobna.

Poniżej zaprezentowano jak zaimportować panel w programie Grafana:

TODO: folissa: scr jak zaimportować

Testowanie oraz omówienie wyników

Niefortunnie dla nas, Jitsi bardzo dobrze się skaluje. Według oficjalnych pomiarów:

- Dla 1056 strumieni wideo z bitrate 550mbit/s zużycie CPU to tylko 20% przy czterordzeniowym procesorze,
- Dla 1056 strumieni wideo Zużycie RAMu nie przekroczyło 1.5GB.

Tworzy to problem stworzenia prawdziwego testu, ponieważ nie posiadamy zasobów aby włączyć tyle strumieni z taką przepustowością. Nie mniej, spróbowaliśmy zbadać jak najdokładniej co dzieje się, kiedy Jitsi jest obciążone.

Wykorzystaliśmy poniższy skrypt do testowania różnych scenariuszy:

```
const puppeteer = require('puppeteer');
const JITSI_URL = 'https://jitsi.google.sex.pl/abc';
const BOT_COUNT = 3;
const DURATION_SECONDS = 300;
const HEADLESS = false;
async function startBot(id) {
    const browser = await puppeteer.launch({
        headless: HEADLESS ? "new" : false,
        args: [
            '--use-fake-ui-for-media-stream',
            '--use-fake-device-for-media-stream',
            '--no-sandbox',
            '--disable-setuid-sandbox',
            '--window-size=1280,720'
        ]
    });
    const page = await browser.newPage();
    await page.setViewport({ width: 1280, height: 720 });
    try {
        await page.goto(`${JITSI_URL}`, { waitUntil: 'networkidle0' });
        try {
            const nameInputSelector = 'input[placeholder="Enter your name"]';
            await page.waitForSelector(nameInputSelector, { timeout: 5000 });
            await page.type(nameInputSelector, `Bot-${id}`);
            const joinButtonSelector = '[data-testid="prejoin.joinMeeting"]';
            await page.waitForSelector(joinButtonSelector);
            await page.click(joinButtonSelector);
        } catch (error) {
            console.log(`Bot ${id}: No prejoin screen detected (or timed out)`);
        }
        await new Promise(r => setTimeout(r, DURATION_SECONDS * 1000));
    } catch (e) {
        console.error(`Bot ${id} FAILED:`, e.message);
    } finally {
        await browser.close();
    }
}
(async () => {
    const promises = [];
    for (let i = 0; i < BOT_COUNT; i++) {
        promises.push(startBot(i));
        await new Promise(r => setTimeout(r, 2000));
    }
    await Promise.all(promises);
})();
```

Listing 5: Skrypt do testowania Jitsi

Zmienialiśmy stałe oraz URL do testowania różnych sytuacji. Aby nie pisać za każdym razem każdych parametrów, należy założyć, że każda wideokonferencja to zmieniony zasób w zmiennej JITSI_URL, oraz różna liczba uczestników to zmienna BOT_COUNT-1 (gdzie dodatkowa osoba to my jako uczestnik).

Wyniki oraz spostrzeżenia sprzed optymalizacji za pomocą Octo

Wyniki oraz spostrzeżenia po optymalizacji za pomocą Octo