

Wdrożenie Jitsi na platformie Kubernetes, zbadanie wydajności oraz próba optymalizacji

Dokumentacja projektu

Jakub Jędrzejczyk

Sebastian Kwaśniak

Anna Berkowska

2025-12-13

Spis treści

1	Wstęp teoretyczny	2
1.1	Kubernetes	2
1.2	Jitsi	2
1.3	Wybór narzędzi	3
2	Scenariusz - projekt	4
2.1	Sieć	4
2.2	Zebranie danych	4
3	Replikacja środowiska	5
3.1	Maszyny	5
3.2	Kubernetes	5
3.3	Jitsi	6
3.4	Monitoring	6
4	Testowanie oraz omówienie wyników	12
4.1	Scenariusze testowania	14
4.2	Wyniki oraz spostrzeżenia przed optymalizacją za pomocą Octo	14
4.3	Wyniki oraz spostrzeżenia po optymalizacji za pomocą Octo	17
5	Wnioski	23

Spis treści

1 Wstęp teoretyczny

1.1 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 uczyści klaster bezpieczniejszym. Niektóre z obiektów mogą być tylko klastrowe, a niektóre 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.

1.1.1 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) wewnętrz 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.

1.2 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 dotyczy 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.

1.2.1 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 dowiedzieliśmy o jitsi octo TODO: doppie: opisac to co sie dowiedzieliśmy o jitsi octo

1.3 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 interakcji 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

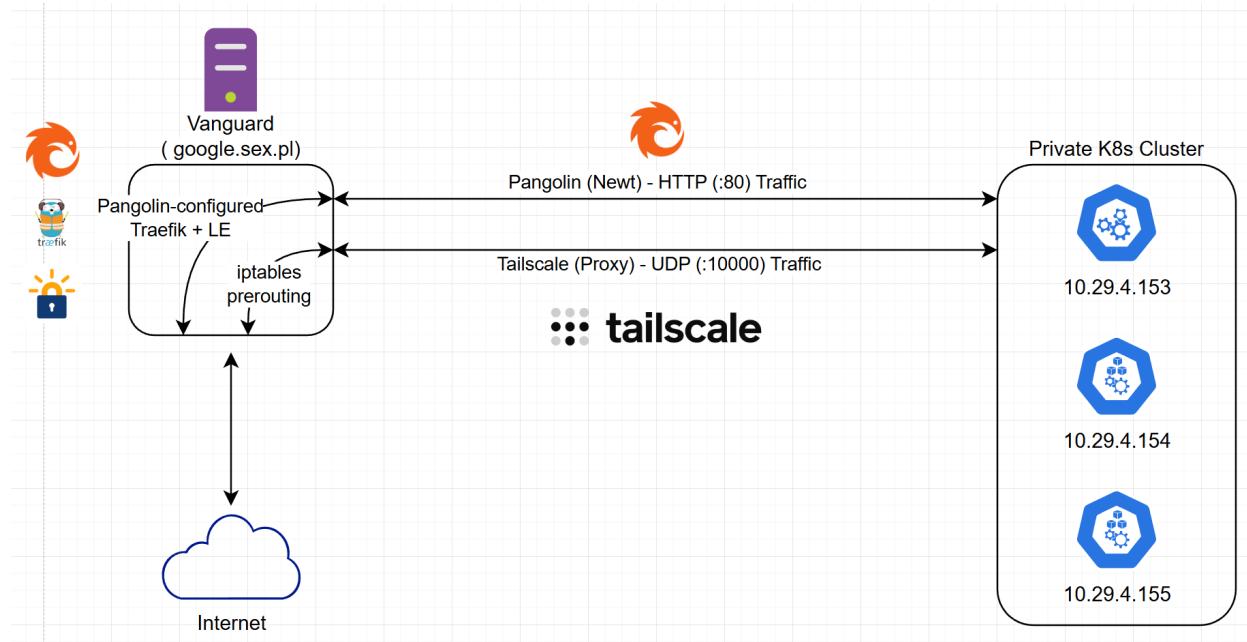
Do prostszego dołączenia wielu użytkowników skorzystamy z platformy node.js oraz dostępnej na niej biblioteki puppeteer.

2 Scenariusz - projekt

2.1 Sieć

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

Sieć została zwizualizowana na rysunku 1.

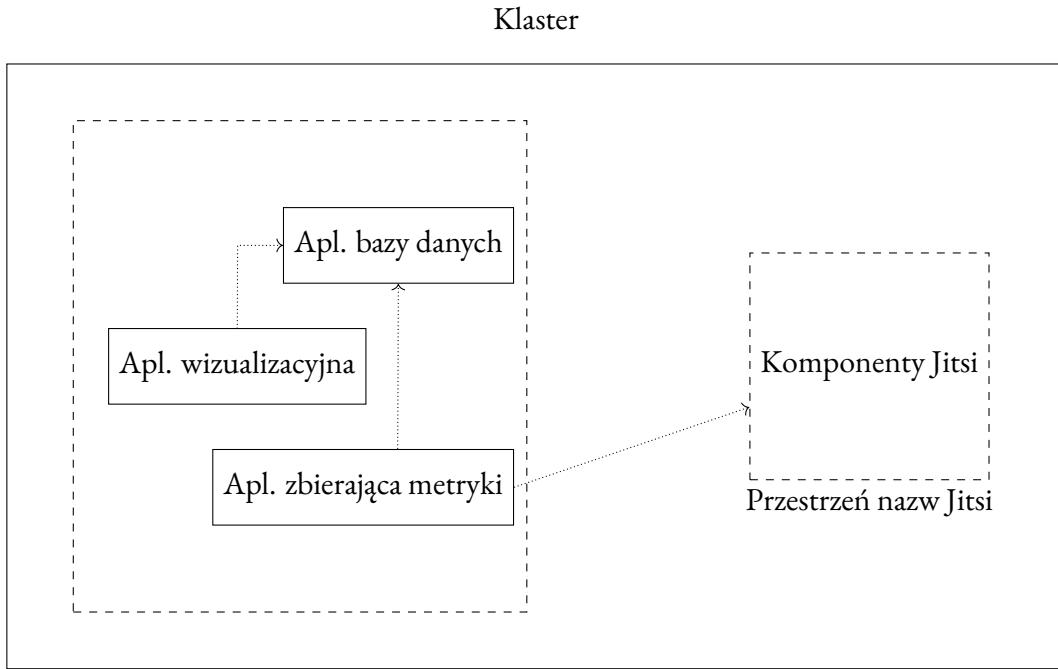


Rysunek 1: Projekt sieci do udostępnienia Jitsi publicznie

2.2 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

3 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.

3.1 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.

3.2 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 folderze 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.

3.3 Jitsi

TODO: doppie:

3.3.1 Udostępnienie Jitsi publicznie

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

3.3.2 Konfiguracja Jitsi z Octo

Do użycia Octo musimy zmodyfikować jego domyślne values.yaml poprzez dopisanie TODO: folissa:

Jednym z wymogów Octo jest posiadanie jakiegoś tam ip TODO: folissa: oraz nie może być na tym samym węźle coś tam coś TODO: folissa

3.4 Monitoring

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

3.4.1 Baza danych

Do przechowywania metryk został wybrany program Mimir w trybie mikroserwisów. Wybrany został tryb wdrożenia, gdzie Mimir rozbity jest na wiele mikroserwisów, ponieważ dobrze działa ona na Kubernetes. Istnieje także architektura, gdzie Mimir zamknięty jest w jednym pliku binarnym. Mikroserwisy pozwalają m.in. na skalowanie aplikacji w prostszy sposób (skalowanie pojedyńczych komponentów), a także zapewnia odporność na przerwę w pracy, ponieważ pozwala na niedostępność pojedyńczych elementów, a nie całej aplikacji.

Został on skonfigurowany w następujący sposób:

```

---
### Sekcja 1 #####
mimir:
  structuredConfig:
    ingest_storage:
      enabled: false
    ingester:
      push_grpc_method_enabled: true
    common:
      storage:
        backend: "s3"
        s3:
          endpoint: "mimir-minio.svc.cluster.local:9000"
          insecure: true
          access_key_id: "grafana-mimir"
          secret_access_key: "supersecret"
### Sekcja 2 #####
store_gateway:
  zoneAwareReplication:
    enabled: false
  persistentVolume:
    enabled: false
  ingester:
    zoneAwareReplication:
      enabled: false
    persistentVolume:
      enabled: false
  compactor:
    persistentVolume:
      enabled: false
  ruler:
    persistentVolume:
      enabled: false
minio:
  persistence:
    enabled: false
### Sekcja 3 #####
kafka:
  enabled: false
alertmanager:
  enabled: false
nginx:
  enabled: false
rollout_operator:
  enabled: false

```

Listing 1: Plik values.yaml programu Mimir

Idąc od góry:

1. W pierwszej sekcji ustawiona jest konfiguracja samego Mimir. Wyłączony jest ingest_storage, a dodatkowo ingester (jeden z komponentów) przyjmuje push_grpc_method_enabled: true, ponieważ nie chcemy używać architektury Mimir 3.0, która używa Kafki. W storage jest konfiguracja kubełków S3, w których przechowywane są metryki, alerty itd. storage ustawiony jest w common, dzięki czemu wszystkie Pod otrzymują tą wspólną konfigurację.
2. W sekcji drugiej wymienione są używane mikroserwisy Mimir np. compactor, ruler. Wyłączone mają one persistence, czy też persistentVolume, ponieważ w przypadku testowania niepotrzebne jest nam długotrwałe przechowywanie metryk. Dane są utracone przy restarcie kontenera. zone-AwareReplication jest również wyłączone, ponieważ mamy jeden kластer składający się z trzech maszyn wirtualnych i nie jesteśmy w stanie rozłożyć aplikacji na więcej klastrów, czy fizycznych lokalizacji, które byłyby tzw. failure domain.
3. W sekcji trzeciej wyłączone są komponenty Mimir, które nie są nam potrzebne do naszego przypadku użycia.

Wdrożenie Mimir następuje za pomocą komend:

```
$ helm repo add grafana https://grafana.github.io/helm-charts  
$ helm install mimir grafana/mimir-distributed --namespace mimir --values values.yaml
```

Komendy robią następująco:

1. Dodaje repozytorium grafana, z którego pobrany zostanie chart
2. Instaluje instancje aplikacji mimir, używając chart grafana/mimir-distributed w przestrzeni nazw mimir, o wartościach values.yaml

3.4.2 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
```

3.4.3 Aplikacja prezentowania danych

Do prezentowania zbieranych danych wykorzystywany jest program Grafana. Skonfigurowana została ona w następujący sposób:

```
---
```

```
adminUser: "admin"  
adminPassword: "admin"
```

Listing 4: Plik values.yaml programu Grafana

Powyższa konfiguracja zawiera plik konfiguracyjny Grafany values.yaml, który konfiguruje użytkownika admin o haśle admin, co w wersji produkcyjnej powinno być schowane w sekrecie, ale dla celów testowych nie ma to większego znaczenia.

Wdrożenie następuje za pomocą komend:

```
$ helm repo add grafana https://grafana.github.io/helm-charts  
$ helm install grafana grafana/grafana --namespace grafana --values values.yaml
```

Komendy robią następująco:

1. Dodaje repozytorium grafana, z którego pobrany zostanie chart (repozytorium wystarczy dodać raz, więc jeżeli było to zrobione wcześniej, to można pominąć)
2. Instaluje instancje aplikacji grafana, używając chart grafana/grafana w przestrzeni nazw grafana, o wartościach values.yaml

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

```

$ export POD_NAME=$(kubectl get pods --namespace grafana \
-l "app.kubernetes.io/name=grafana,app.kubernetes.io/instance=grafana" \
-o jsonpath="{.items[0].metadata.name}")
$ kubectl --namespace grafana port-forward $POD_NAME 3000

```

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

Przygotowany został panel przedstawiający zebrane metryki z Jitsi. Poniżej zaprezentowano jak zaimportować panel w programie Grafana:

TODO: folissa/sebsk - podać ten panel na enauczanie i opisać tutaj

Aby zaimportować panel, wystarczy wejść do Grafana w zakładkę “Dashboards” oraz wybrać “New” po czym przejść do opcji “Import”.

Po przejściu do opcji “Import” wystarczy wkleić dołączony do dokumentacji panel oraz wcisnąć “Load”.

4 Testowanie oraz omówienie wyników

Twórcy Jitsi utrzymują, że ich program 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 (gdzie dodatkowa osoba to my jako uczestnik).

4.1 Scenariusze testowania

Aby sensownie porównać przed i po przeskalowaniu, musimy stwierdzić jakie scenariusze testowania są odpowiednie. W każdym ze scenariuszy zostanie przesłany źródłowy strumień wideo w rozdzielczości 1280x720. Są to kolejno:

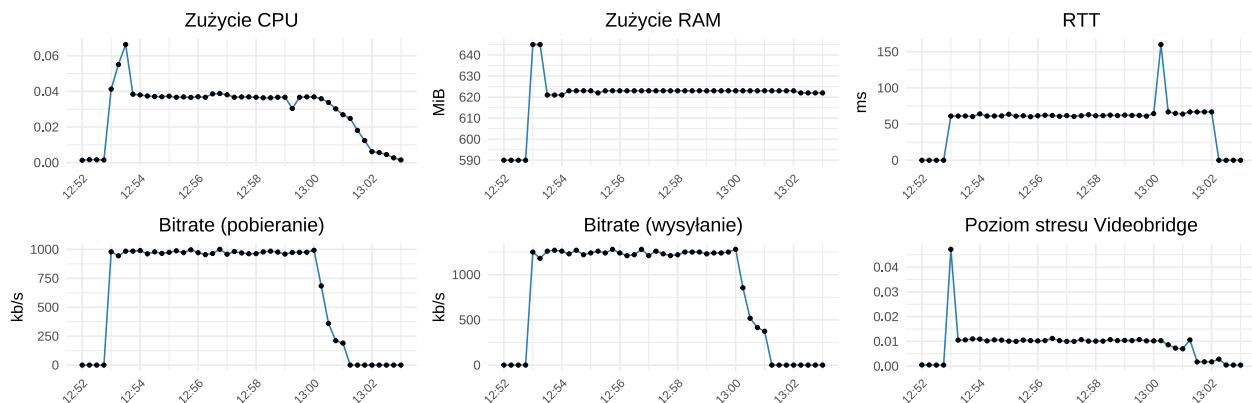
1. 1 konferencja 3 uczestników - testowa konferencja,
2. 1 konferencja 10 uczestników - przy porównaniu z 1 konferencją i trzema uczestnikami uzyskamy sens wielkości w jakich się poruszamy,
3. 3 konferencje po 10 uczestników - sprawdzenie możliwości, w przypadku optymalizacji przetestowanie gdzie zostaną umieszczeni użytkownicy końcowi,
4. 10 konferencji po 3 uczestników - do finalnego dokładniejsze sprawdzenia umieszczenia, w wypadku testowania po optymalizacji.

Odnośnie 3 uczestników, jest to najprostszy sposób na wymuszenie Jitsi aby konferencja nie była p2p, przy testowaniu z jednego źródła. Cały ruch wtedy odbywa się bezpośrednio przez JVB.

Przed pójściem dalej, warto zaznajomić się z wartościami które nie są oczywiste:

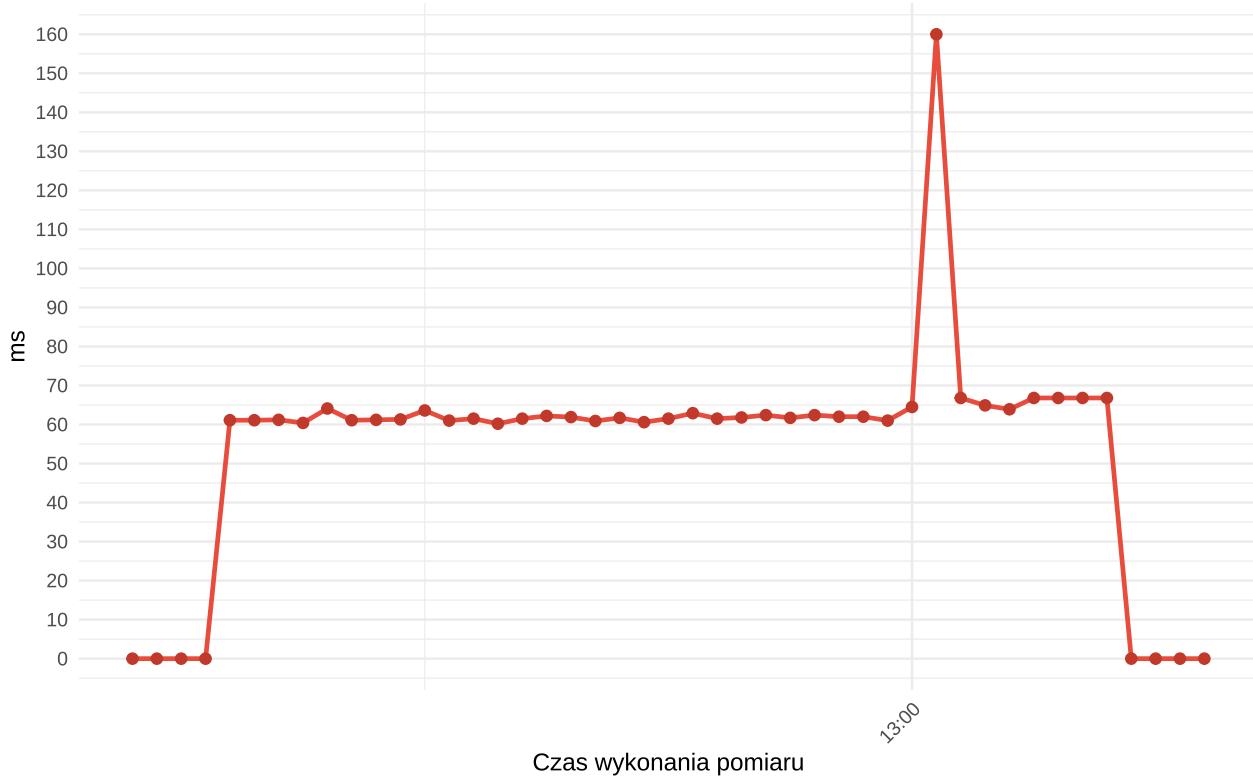
- zużycie CPU - w Kubernetes wartość CPU nie jest podawana w procentach całego serwera, lecz w jednostkach rdzeni logicznych (vCPU lub Core),
- poziom stresu Videobridge - całkowity obraz jak wykorzystywany jest Videobridge, gdzie 0 oznacza brak obciążenia, a 1 pełne wykorzystanie zasobów (chociaż wartości większe niż 1 są dopuszczalne).

4.2 Wyniki oraz spostrzeżenia sprzed optymalizacji za pomocą Octo



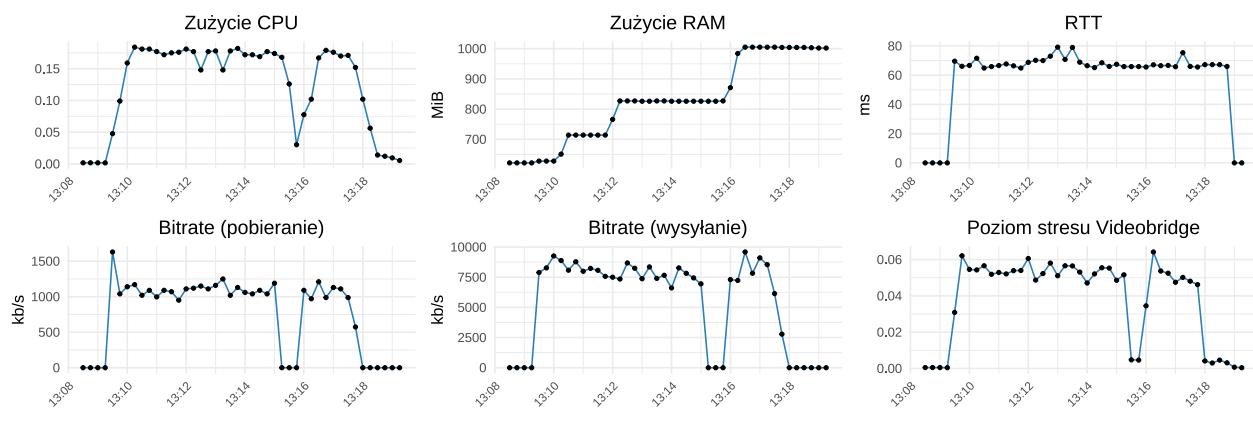
Rysunek 3: Scenariusz 1 - 1 konferencja, 3 uczestników

Przy pierwszym scenariuszu widzimy dużą korelację pomiędzy zużyciem procesora, RAMu oraz poziomu stresu VideoBridge. Widzimy także, że początek konferencji jest najbardziej wymagający dla całego systemu. Przejrzyjmy się bliżej metryce RTT.



Rysunek 4: Szczegółowy wykres RTT dla scenariusza 1

W tym scenariuszu RTT występuje pomiędzy 60 a 70 milisekund. Jest to zadowalający wynik, patrząc na to, że architektura sieciowa tej aplikacji jest naprawdę skomplikowana. Wystąpiła jedna potocznie nazywana "czkawka" (hiccup), w której RTT wynosiło 160ms. Ta "czkawka" występuje z każdym z następnych pomiarów, które nie były zoptymalizowane za pomocą octo.

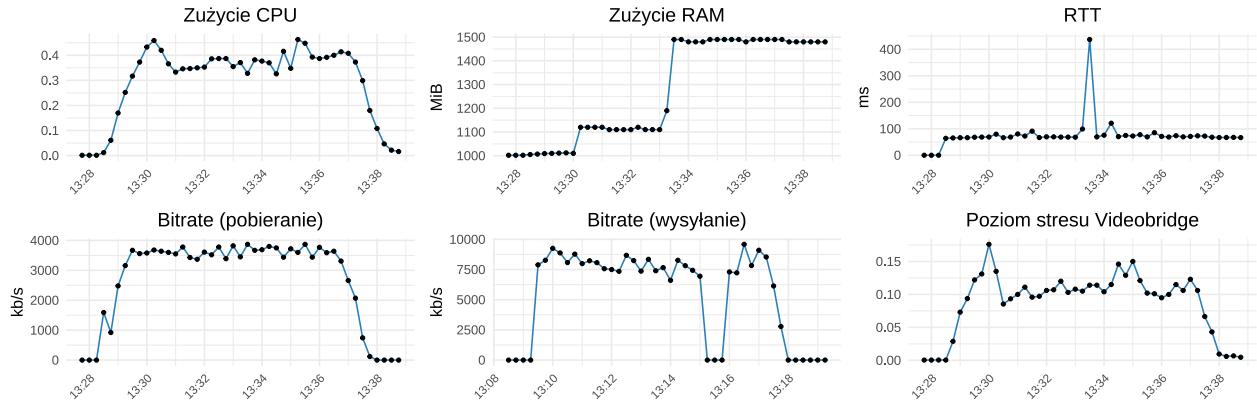


Rysunek 5: Scenariusz 2 - 1 konferencja, 10 uczestników

Przy scenariuszu 2 pierwsze rzeczy jakie rzucają się w oczy to nieoczekiwany spadek zużycia CPU oraz zerowy bitrate około godziny 13:16. Przez charakter testu, nie wiemy co dokładnie się wydarzyło, lecz Jitsi

działało poprawnie. Zakładamy, że był to błąd systemu którego zadaniem było testowanie Jitsi. Nie mniej, widać korelację między każdym z elementów tak, jak miało miejsce w poprzednim pomiarze.

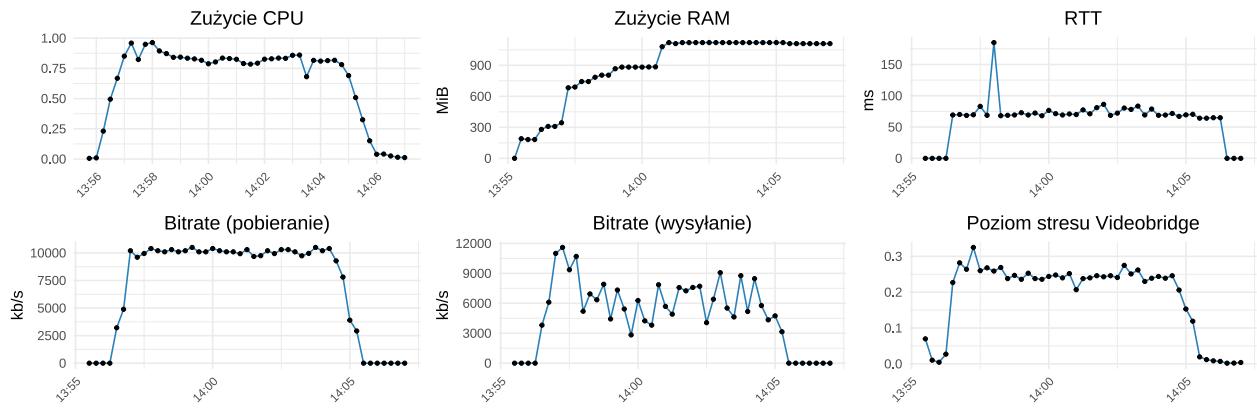
Warto zauważyc, że zużycie RAMu jest wysokie i nie spada - taka sytuacja dzieje się także na następnym pomiarze.



Rysunek 6: Scenariusz 3 - 3 konferencje po 10 uczestników

Tu widzimy, że zużycie RAMu przekracza w tym momencie 1.5GB, co nie zgadza się z pomiarami zaprezentowanymi przez twórców Jitsi. Nawet jeśli założymy najbardziej pozytywny dla ich pomiarów scenariusz (niewygaśnięcie poprzednich rozmów, utrzymywanie wszystkich łącz) mamy nadal sytuację, gdzie istnieją 43 strumienie wideo oraz audio, co jest ponad 20-krotnie mniejszą liczbą niż deklarowane 1053 przy takim zużyciu zasobów.

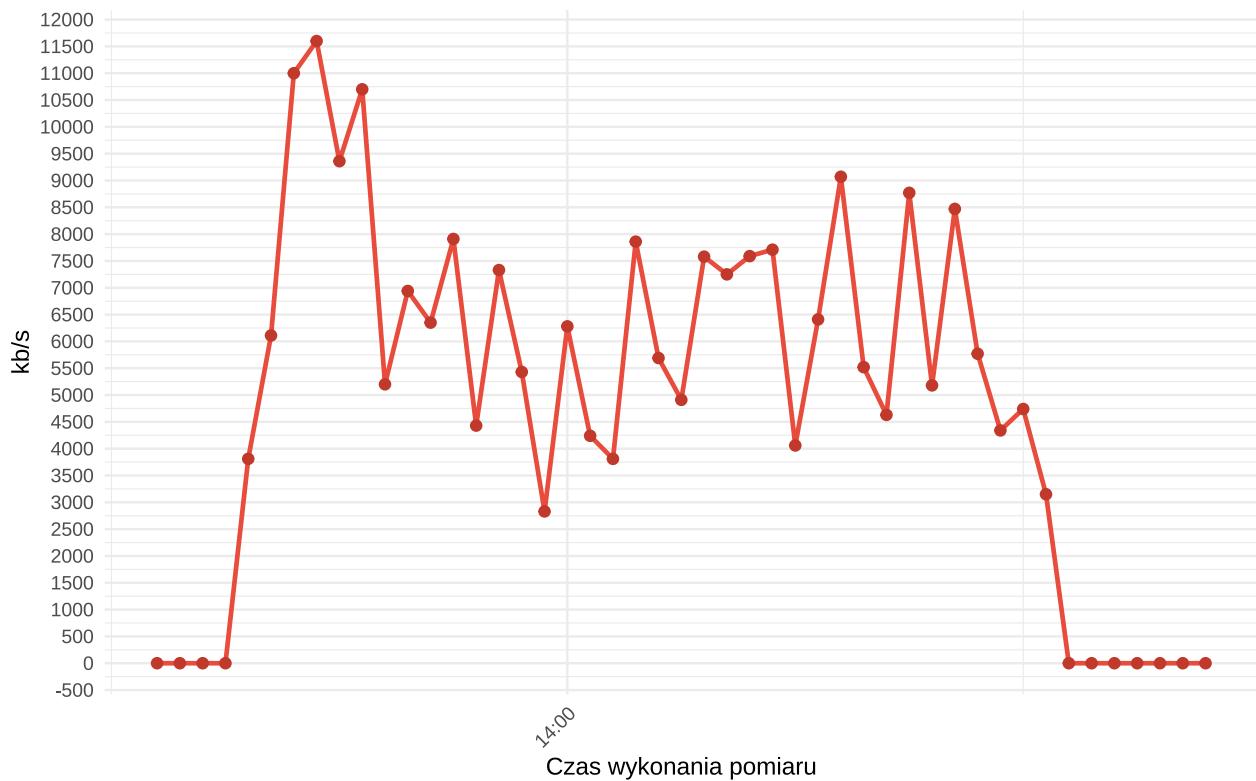
Przy tych wykresach warto zapamiętać zużycie CPU na poziomie 0.4, poziom stresu na poziomie 0.1 oraz bitrate na poziomie 4000 (pobieranie), przydadzą się do następnych wniosków.

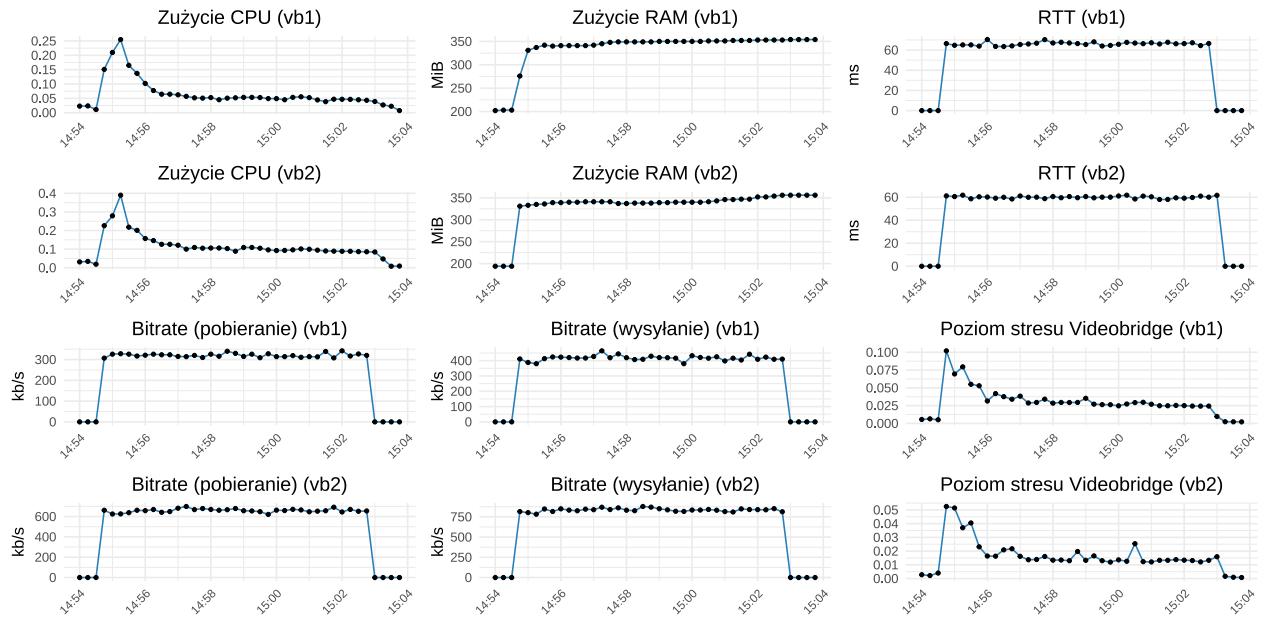


Rysunek 7: Scenariusz 4 - 30 konferencji po 3 uczestników

Tu jasno widać pięć achillesową Jitsi - ilość konferencji. Przy sytuacji tej samej ilości uczestników, lecz wyższej liczbie konferencji system potrzebuje lekko ponad dwukrotności zasobów. Zużycie CPU jest na poziomie 1.0, a bitrate pobierania jest na poziomie 10000. Poziom stresu także zwiększył się podwojnie, zgodnie z korelacją którą zaznaczyliśmy w wynikach z pierwszego scenariusza.

Tutaj także widzimy najbardziej “poszarpany” wykres wysyłania bitrate.





Rysunek 9: Scenariusz 1 - 1 konferencja, 3 uczestników

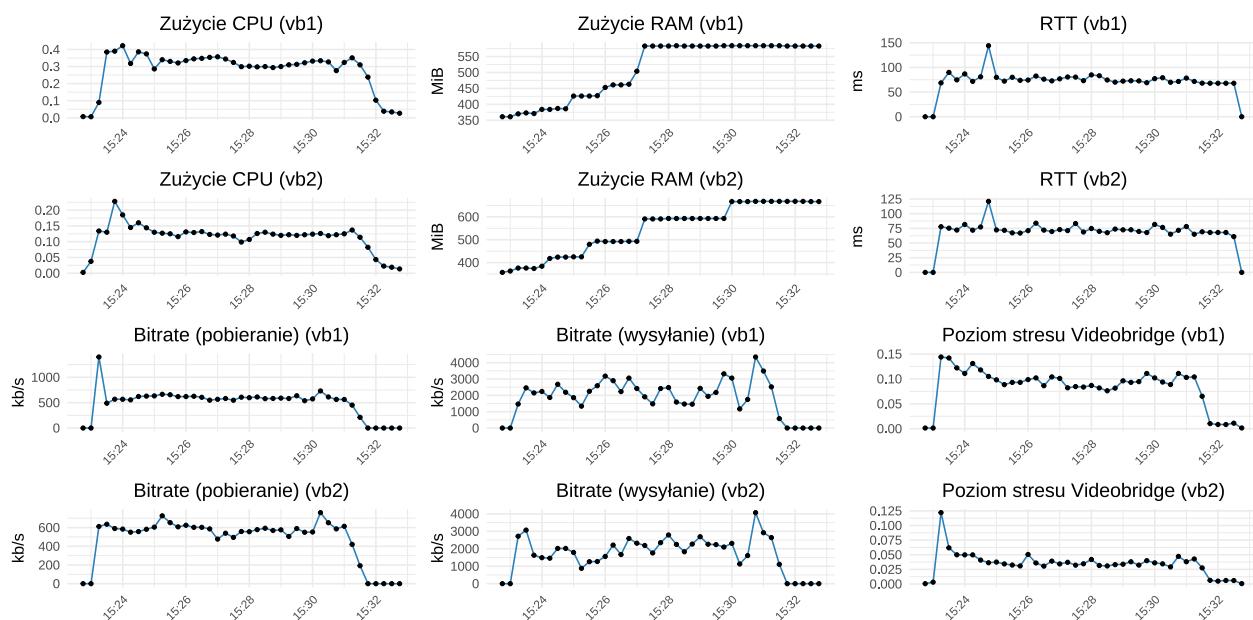
Już na starcie widać korelację między obydwojoma instancjami. Zużycie wzrasta na obydwu równomiernie, lecz vb1 konsekwentnie jest o połowę mniej obciążone niż vb2. Jak poprzednio, korelacja z poziomem stresu jest taka sama. Przyjrzyjmy się RTT dla obydwu sytuacji.





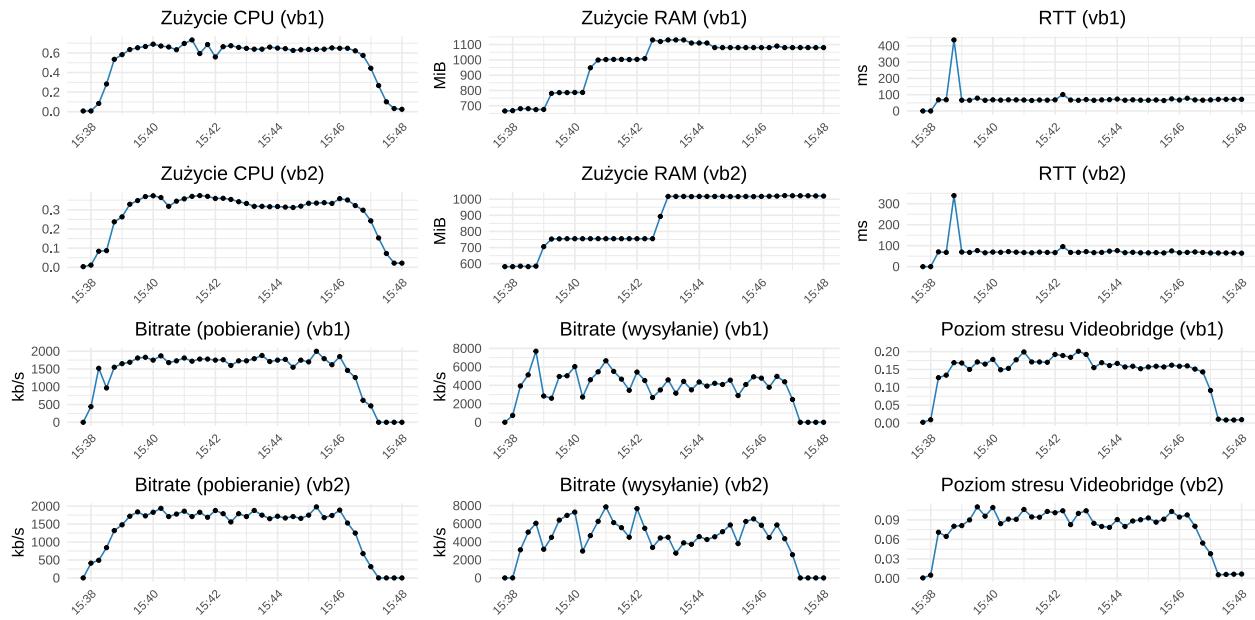
Rysunek 11: Scenariusz 1 OCTO - RTT (vb2)

Można zauważyc, że mniej obciążony Videobridge ma delikatnie większy RTT (różnica ok. 5ms). Jak wspomniane wcześniej, wnioski przedstawione będą na końcu.



Rysunek 12: Scenariusz 2 - 1 konferencja, 10 uczestników

Dla scenariusza drugiego vb1 został obciążony bardziej. W RTT znów widzimy nagły skok. Bitrate dla wysyłania jest tutaj poszarpany o wiele bardziej.



Rysunek 13: Scenariusz 4 - 3 konferencje po 10 uczestników

Dla scenariusza 3 jest najwyższy skok RTT. Poniżej przybliżenie wykresów.

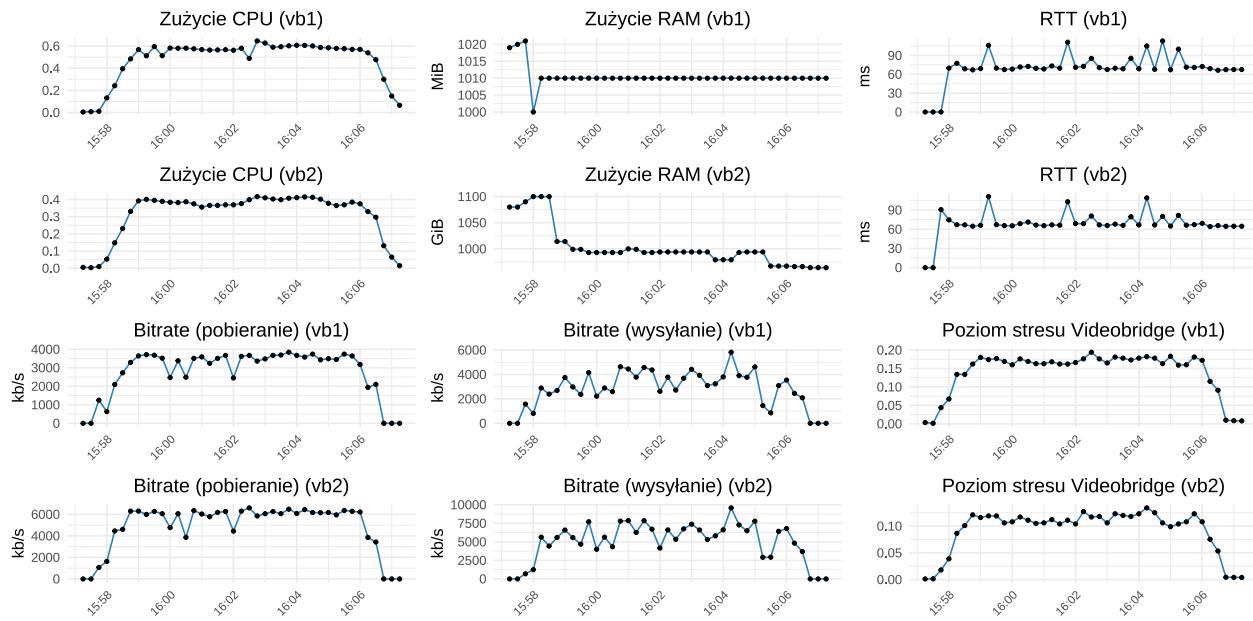


Rysunek 14: Scenariusz 1 OCTO - RTT (vb1)



Rysunek 15: Scenariusz 1 OCTO - RTT (vb2)

Fortunnie, średnio RTT wynosi około 75ms. Ten jeden skok nie miał dużego wpływu na rozmowę.



Rysunek 16: Scenariusz 4 - 30 konferencji po 3 uczestników

W scenariuszu 4 znów vb2 jest obciążone bardziej. Dodatkowo widzimy niespodziewaną sytuację: zużycie RAMu spadło dla obydwu instancji. Inną obserwacją jest stałe zużycie RAMu po spadku przy vb2, które ciągnie się do końca transmisji.

5 Wnioski

Głównym, najbardziej nieoczekiwany i niezrozumianym spostrzeżeniem jest nieprzewidywalne zużycie pamięci RAM. Rośnie ono z każdą następną konferencją, oraz maleje w różnych momentach. Według nas, jest spowodowane to przetrzymywaniem pokoi konferencyjnych po ich zakończeniu, aby umożliwić szybki powrót.

Drugą rzeczą jest różnica 5 milisekund w RTT gdy patrzymy na konferencje w OCTO. Według nas, jest spowodowane to dodatkowym przesyłem danych pomiędzy instancjami videobridge.

Widać jednoznaczną korelację z ilością konferencji a znacząco wyższym bitrate. Bardziej obciążające łącze jest wiele konferencji przy tej samej ogólnej ilości uczestników. Fortunnie, w sytuacji octo widzimy rozłożenie tego obciążenia.

Jedną z obserwacji która nie jest udokumentowana, była dynamiczna zmiana rozdzielczości przez Jitsi aby utrzymać akceptowalny poziom jakości rozmowy. Widzieliśmy skok rozdzielczości z 1280x720 do 360x180, gdy pojawiało się więcej uczestników. Nasze testy, mimo bycia konkretnymi scenariuszami, nadal mogą być cięzsze w dokładnym odtworzeniu, ze względu na dynamiczne zachowanie programu. Przy przybliżeniu w aplikacji jednego z uczestników, który przesyła strumień wideo, rozdzielczość oraz bitrate dla tego wideo jest automatycznie zwiększone. W rysunku 8 został ukazany najmniej zrozumiany przez nas wykres. Wydaje nam się, że przez wysokie zużycie Jitsi, próbowało ono dynamicznie zmniejszać oraz zwiększać rozdzielczość tak, aby konsekwentnie rozdzielić swoje zasoby pomiędzy wszystkie konferencje.