

Wdrożenie Jitsi na platformie Kubernetes, zbadanie wydajności oraz próba optymalizacji

Dokumentacja projektu

Jakub Jędrzejczyk

Sebastian Kwaśniak

Anna Berkowska

2025-12-19

Spis treści

1 Wstęp teoretyczny	2
1.1 Kubernetes	2
1.2 Jitsi	2
1.3 Wybór narzędzi	5
2 Scenariusz - projekt	6
2.1 Sieć	6
2.2 Zebranie danych	7
3 Replikacja środowiska	8
3.1 Maszyny	8
3.2 Kubernetes	8
3.3 coturn	9
3.4 Jitsi	10
3.5 Monitoring	22
4 Testowanie oraz omówienie wyników	28
4.1 Scenariusze testowania	30
4.2 Wyniki oraz spostrzeżenia przed optymalizacją za pomocą Octo	30
4.3 Wyniki oraz spostrzeżenia po optymalizacji za pomocą Octo	33
5 Wnioski	39

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:

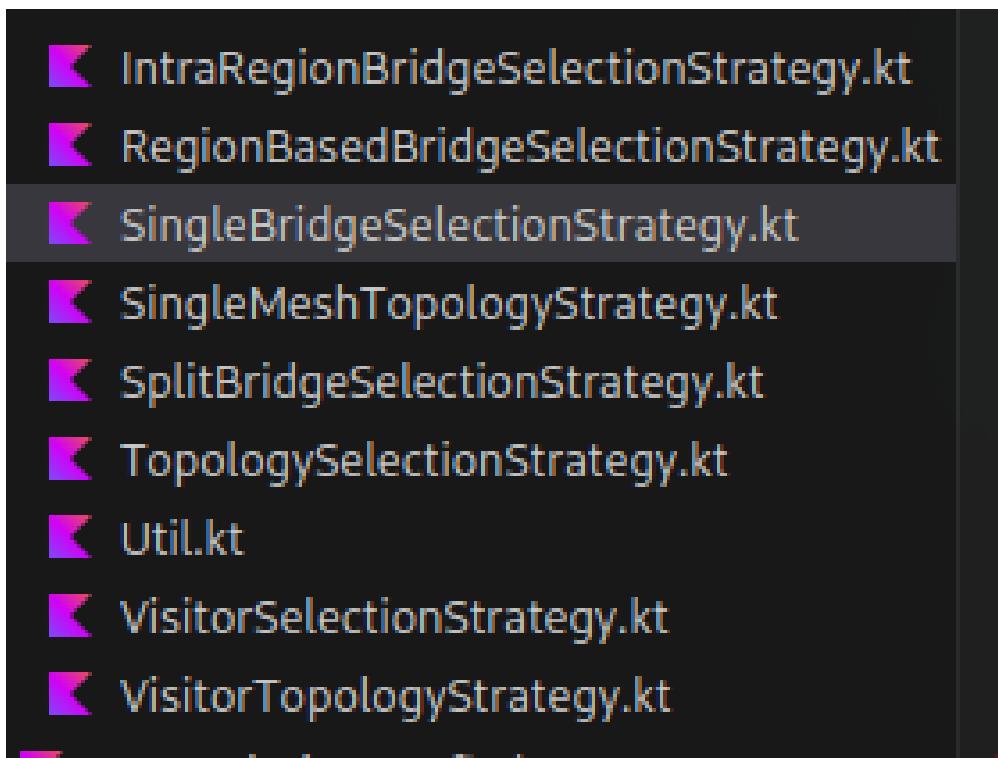
- JVB (Jitsi Videobridge) - JVB odbiera strumienie audio/wideo od użytkowników i przesyła je dalej.
 - Simulcast: JVB dba o jakość - przy słabym łączu JVB wysyła strumień o niższej jakości, nie obciążając przy tym nadawcy.
- Jicofo (Jitsi Conference Focus) - Zarządza sesjami konferencyjnymi.
 - Działanie: 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.

Po włączeniu Octo, Jicofo rozdziela spotkania między różne mostki JVB na podstawie skonfigurowanej “strategii wyboru mostka” (z ang. “*bridge selection strategy*”). Dostępnych strategii jest kilka, a domyślna w używanej przez nas konfiguracji to “RegionBasedBridgeSelectionStrategy”, która stara się łączyć użytkowników z mostkami w ich regionie geograficznym, jednocześnie pozwalając na pewne odchylenia w celu równoważenia obciążenia. Pozostałe (z wyjątkiem pliku Util.kt który zawiera funkcje pomocnicze) są widoczne na rysunku 1.



Rysunek 1: Dostępne strategie wyboru mostka w Jitsi Octo

Kod odpowiedzialny za wybór mostka w przypadku tej funkcji składa się z wykorzystania kilku funkcji pomocniczych, które starają się znaleźć najlepszy mostek dla użytkownika - funkcje te są również wykorzystane w innych metodach wyboru mostka, jednak te nie zostały przez nas przetestowane.

```
return notLoadedAlreadyInConferenceInRegion(bridges, conferenceBridges,
    ↵ participantProperties, region)
    ?: notLoadedAlreadyInConferenceInRegionGroup(bridges, conferenceBridges,
        ↵ participantProperties, region)
    ?: notLoadedInRegion(bridges, conferenceBridges, participantProperties, region)
    ?: notLoadedInRegionGroup(bridges, conferenceBridges, participantProperties,
        ↵ region)
    ?: notLoadedAlreadyInConference(bridges, conferenceBridges,
        ↵ participantProperties)
    ?: notLoaded(bridges, conferenceBridges, participantProperties)
    ?: leastLoadedNotMaxedAlreadyInConference(bridges, conferenceBridges,
        ↵ participantProperties)
    ?: leastLoaded(bridges, conferenceBridges, participantProperties)
```

Listing 1: Fragment kodu strategii RegionBasedBridgeSelectionStrategy

Powyższy kod stara się znaleźć najmniej obciążony mostek dla użytkownika, stosując się do następującej kolejności:

- Nieprzeładowany mostek w konferencji w danym regionie (lub jego grupie),
- Nieprzeładowany mostek w danym regionie (lub jego grupie), lecz niekoniecznie w konferencji,
- Nieprzeładowany mostek w regionie (lub jego grupie), kiedy konferencja ma mostki w regionie, lecz wszystkie są przeładowane,
- Najmniej przeładowany mostek znajdujący się w grupie regionu użytkownika oraz konferencji,
- Najmniej przeładowany mostek znajdujący się w grupie regionu użytkownika,
- Nieprzeładowany mostek znajdujący się w konferencji,
- Najmniej przeładowany mostek.

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 klawiszu.

Programy które wdrożymy na klawiszu 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 wizualizacjistępnej (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:

- Pangolin - oparty na WireGuard zestaw aplikacji umożliwiający łatwe udostępnianie usług po HTTP(s),
- Tailscale - oparty na WireGuard mesh'owy VPN pozwalający na łatwe łączenie urządzeń i tworzenie "wirtualnej sieci" między nimi,
- Coturn - serwer TURN oraz STUN umożliwiający przekazywanie ruchu multimedialnego przez NATy i firewalle.

Dwa z powyższych rozwiązań (Pangolin oraz Tailscale) zostały wybrane ze względu na fakt, iż były już wykorzystywane przez jedną osobę z zespołu. Dodatkowo wykorzystanie zostaną serwer należący do jednego członka zespołu mający publiczny adres IPv4, na którym postawione są wyżej wymienione aplikacje.

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

2 Scenariusz - projekt

2.1 Sieć

Postawienie Jitsi tak, aby było publicznie dostępne, wymaga otwarcia kilku portów (wg. strony: <https://meetrix.io/blog/webrtc/jitsi/meet/what-port-your-should-open.html>): 443 dla HTTPS, 10000 UDP dla ruchu multimedialnego, oraz 4443 TCP jako alternatywa dla 10000 UDP. Port 443, tj. HTTPS, jest niezbędny do częściowej Jitsi, a także do sygnalizacji WebRTC. Port 10000 UDP jest używany do przesyłania strumieni audio i wideo między klientami a serwerem Jitsi Videobridge. Port 4443 TCP służy jako alternatywna ścieżka dla ruchu multimedialnego, gdy port 10000 UDP jest zablokowany lub niedostępny.

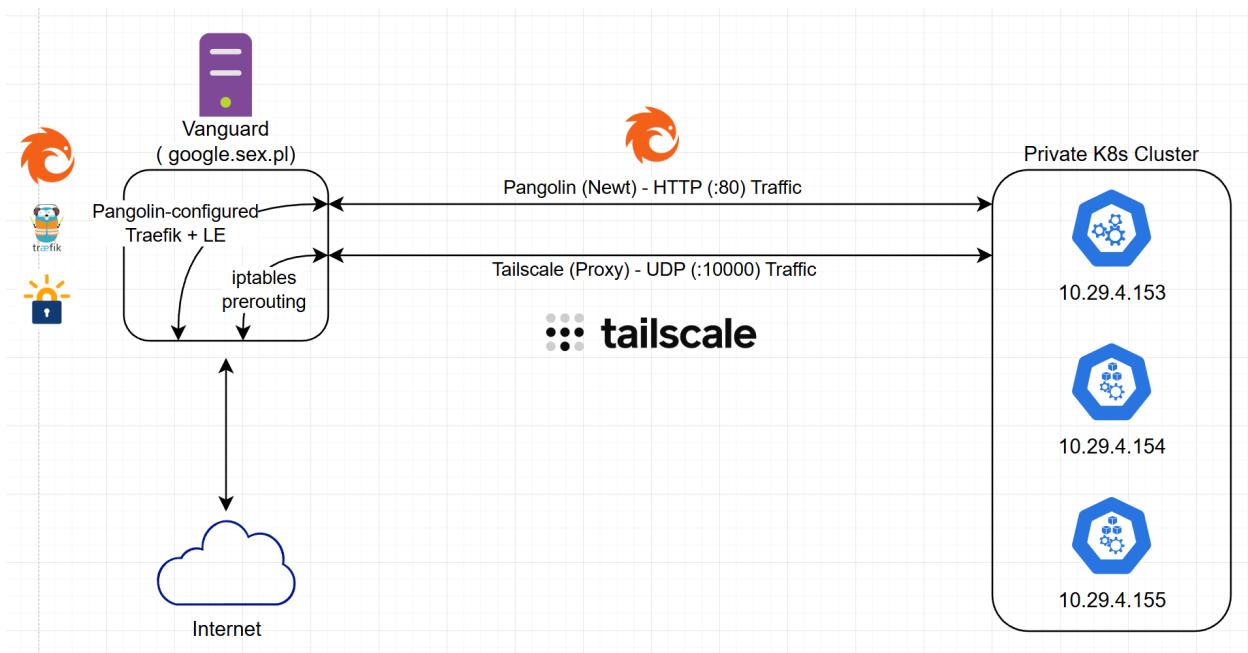
Ze względu na ograniczenia otrzymanego środowiska (brak publicznego adresu IP), użyty zostanie serwer jednego z członków zespołu, który posiada publiczny adres IP i już służy do udostępniania usług w sieci publicznej.

Udostępnienie portu 443 jest możliwe dzięki Pangolinowi, który odpowiednio przekieruje ruch wysyłany pod odpowiedni adres URL do klastra Kubernetes. Pangolin likwiduje również potrzebę na ręczne zarządzanie certyfikatami TLS dla Jitsi, ponieważ automatycznie generuje i odnawia certyfikaty Let's Encrypt dla skonfigurowanych domen.

W teorii porty 10000 UDP oraz 4443 TCP mogłyby być również przekierowane przez Pangolina, jednakże jego backend (Traefik) wymaga podawania pełnych domen dla reguł routingu, co jest problematyczne w przypadku gdy część aplikacji wymaga adresu IP w formie numerycznej. Dotychczasowe doświadczenie związane z udostępnianiem usług na "czystych" TCP / UDP przez Pangolina jest znacznie mniej intuicyjne niż w przypadku HTTP(s). Z tego powodu, do przekierowania ruchu multimedialnego zostanie użyty Tailscale, który umożliwia łatwe tworzenie bezpiecznych połączeń VPN między urządzeniami. Dzięki Tailscale, można stworzyć dodatkowy kontener który będzie mieć dostęp jedynie do sieci JVB i będzie przekazywać mu ruch multimedialny przychodzący z publicznego serwera. Po stronie używanego przez nas serwera z publicznym IP wymagane będzie jedynie dodanie krótkiego wpisu w iptables, który przekieruje ruch z portu 10000 na docelowy adres IP sieci Tailscale poda dołączonego do JVB.

Dodatkowo na samym serwerze z publicznym IP zostanie uruchomiony serwer Coturn, który będzie służył jako serwer STUN/TURN dla Jitsi. Jego zadaniem będzie pomóc w nawiązywaniu połączeń WebRTC, szczególnie w sytuacjach gdy klienci znajdują się za NATami lub firewallami. Serwer Coturn będzie skonfigurowany do obsługi zarówno protokołu UDP, jak i TCP na standardowym porcie 3478, co zapewni maksymalną kompatybilność z różnymi środowiskami sieciowymi użytkowników końcowych. Dzięki temu rozwiązaniu, Jitsi będzie mogło efektywnie obsługiwać połączenia multimedialne nawet w trudnych warunkach sieciowych.

Sieć została zwizualizowana na rysunku 2.

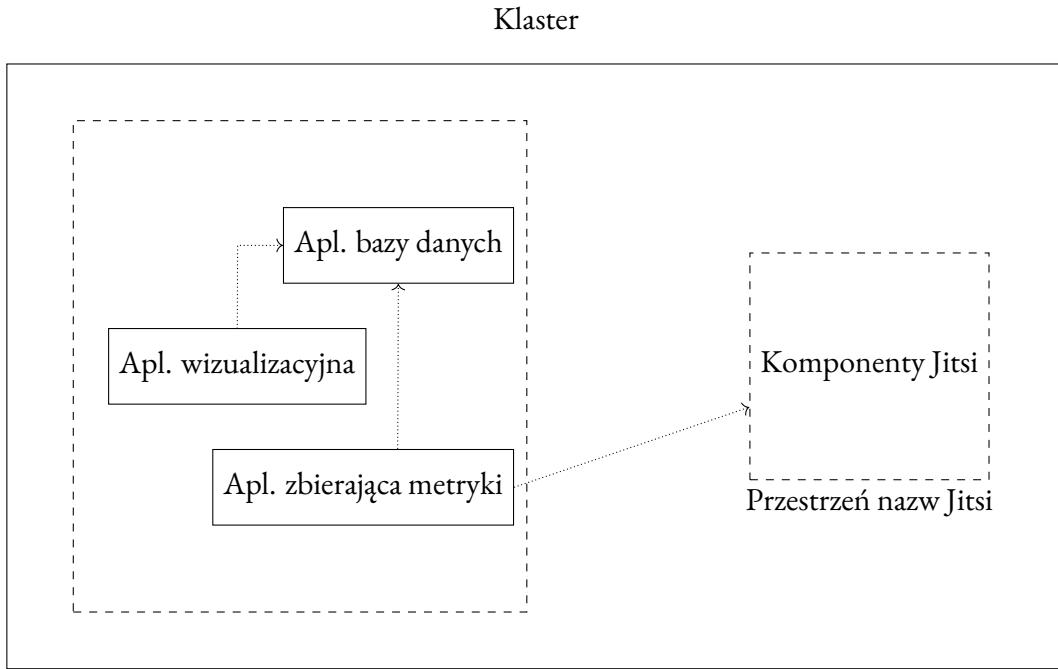


Rysunek 2: Projekt sieci do udostępnienia Jitsi publicznie

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 3.



Rysunek 3: 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 coturn

Serwer coturn zostanie uruchomiony na serwerze z publicznym IP, na którym zostaną otwarte porty 3478 UDP oraz 3478 TCP. Uruchomienie będzie korzystać z zainstalowanego wcześniej Dockera oraz zmodyfikowanego pliku przykładowego dostępnego na stronie GitHub projektu coturn: <https://github.com/coturn/coturn/blob/master/docker/docker-compose-all.yml>, widocznego na rysunku 4.

```
❶ docker-compose.yaml
  ▷ Run All Services
1   services:
  ▷ Run Service
2     coturn:
  ▷ Container Configuration
3       image: coturn/coturn:4.6.3      # pin to the current stable tag
4       container_name: coturn
5       restart: unless-stopped
6       ports:
  ▷ Bind Configuration
7         # STUN/TURN
8         - "3478:3478"
9         - "3478:3478/udp"
10        # TLS-TURN
11        - "5349:5349"
12        - "5349:5349/udp"
13        # RTP/RTCP relays (adjust range if you need fewer ports)
14        - "49000-49020:49000-49020/udp"
15       environment:
  ▷ Secure long-term credentials (generate once; keep secret)
16         STATIC_AUTH_SECRET: "${TURN_STATIC_AUTH_SECRET}"
17         EXTERNAL_IP: "158.101.210.198"
18       volumes:
  ▷ Bind Configuration
19         # Bind your custom conf + persistent database & logs
20         - "/gdzie/tylko/chcesz/coturn/turnserver.conf:/etc/coturn/turnserver.conf:ro"
21         - "coturn-data:/var/lib/coturn"
22         - "coturn-logs:/var/log"
23       # Use host networking if you prefer not to publish individual ports
24       # network_mode: "host"
25
26
27     volumes:
28       coturn-data:
29       coturn-logs:
```

Rysunek 4: Plik docker compose dla serwera coturn

Coturn może korzystać z baz danych do przechowywania użytkowników, jednakże w naszym przypadku skorzystamy z prostszej metody, czyli zdefiniowanie sekretu, którego użytkownik może użyć do uwierzytelnienia się na serwerze.

```

⚙ turnserver.conf
1   realm=google.sex.pl
2   external-ip=158.101.210.198
3   fingerprint
4   lt-cred-mech
5
6   static-auth-secret=abcdefghijklmnopqrstuvwxyz012345
→

```

Rysunek 5: Plik konfiguracyjny serwera coturn

3.4 Jitsi

Po uruchomieniu klastra, posiadając działający kubectl oraz helm, możemy przejść do wdrożenia Jitsi. Ze względu na skorzystanie z serwera coturn, wykorzystany plik values.yaml widoczny na rysunku 6 jest większy niż byłby, gdyby takiego serwera nie było. Linijki 5 oraz 16-23 mogłyby w takim przypadku zostać usunięte.

```

📄 jitsi-values.yaml
1   publicURL: "jitsi.google.sex.pl"
2
3   jvb:
4     useHostPort: true
5     stunServers: 'turn:158.101.210.198:3478?transport=tcp,turn:158.101.210.198:3478?transport=udp'
6     publicIPs:
7       - '158.101.210.198'
8     service:
9       enabled: true
10    type: ClusterIP
11    externalTrafficPolicy: ""
12
13   prosody:
14     transcriber:
15       enabled: false
16     extraEnvFrom:
17       - secretRef:
18         name: jitsi-turn-secret
19
20   web:
21     extraEnvsFrom:
22       - secretRef:
23         name: jitsi-turn-secret

```

Rysunek 6: Plik konfiguracyjny Helm dla Jitsi

Ze względu na konfigurację serwera coturn, która wymaga od użytkownika podania sekretu w celu uwierzytelnienia się, wcześniej wspomniane linijki 16-23 pozwalają serwerowi Jitsi na przekazanie tych danych użytkownikowi, który następnie może ich użyć w celu skorzystania z serwera coturn. W teorii możliwe jest wydobycie sekretu z tej komunikacji, jednakże szacujemy że jest to relatywnie małe zagrożenie (niewiele osób wie o istnieniu usługi, jeszcze mniej raczej potrzebuje serwera coturn), więc jest to akceptowalne ryzyko.

Sam sekret znajduje się w osobnym pliku turn-secret.yaml, którego przykład jest widoczny na rysunku 7. Oprócz niego znajdują się w nim również adres oraz port.

```

turn-secret.yaml
1  apiVersion: v1
2  kind: Secret
3  metadata:
4    name: jitsi-turn-secret
5  type: Opaque
6  stringData:
7    TURN_CREDENTIALS: "abcdefghijklmnopqrstuvwxyz012345"
8    TURN_HOST: "158.101.210.198"
9    TURN_PORT: "3478"

```

Rysunek 7: Plik sekretu dla serwera coturn

Posiadając powyższe dwa pliki, możliwe jest uruchomienie Jitsi przy pomocy dwóch komend, widocznych na rysunku 8. Pierwsza aplikuje sekret, a druga instaluje Jitsi przy pomocy Helma.

```

1 # postawienie samego jitsi
2
3 kubectl apply -f turn-secret.yaml
4 helm install myjitsi jitsi/jitsi-meet --values values.yaml

```

Rysunek 8: Komendy do uruchomienia Jitsi

Tym samym Jitsi uruchamia się na klastrze, lecz jego przetestowanie nie jest możliwe bez udostępnienia go publicznie.

3.4.1 Udostępnienie Jitsi publicznie

W celu instalacji Pangolina, niezbędne jest uprzednio zainstalowanie Dockera na serwerze z publicznym IP. W naszym przypadku serwer jest użytkowany od ponad roku i miał już zainstalowanego Dockera. Trudno jest nam określić konkretnie jaką wersję instalowana była wtedy, lecz ze względu na użytkowanie systemu Ubuntu na serwerze, najpewniej wykorzystana była ówczesna wersja poradnika do instalacji Dockera na Ubuntu, aktualnie dostępna pod adresem: <https://docs.docker.com/engine/install/ubuntu/>.

Instrukcje instalacji Pangolina znajdują się pod adresem: <https://docs.pangolin.net/self-host/quick-install>. Wymagane jest pobranie instalatora, a następnie uruchomienie go z uprawnieniami administracyjnymi:

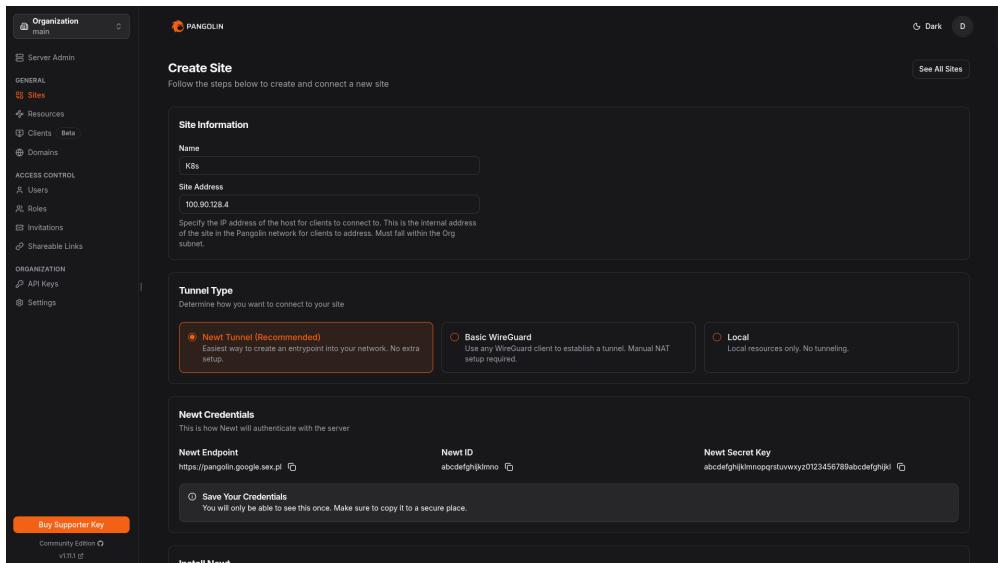
```
curl -fsSL https://static.pangolin.net/get-installer.sh | bash
sudo ./installer
```

Listing 2: Komendy do instalacji Pangolina

Instalator poprosi o podanie wykorzystywanej domeny, adresu pod którym ma być dostępny dashboard (pod wcześniej podaną domeną), email do logowania oraz certyfikatów Let's Encrypt oraz czy tunelowanie powinno być włączone (które należy włączyć aby móc skorzystać z łatwej konfiguracji jaką oferuje Pangolin).

Po kilkuminutowej instalacji należy utworzyć konto administratorskie, po czym możliwe będzie zalogowanie się do dashboardu Pangolina.

Przy użyciu Pangolina, możliwe jest udostępnianie usług na tym samym urządzeniu, na którym jest on zainstalowany, lecz w tym konkretnym przypadku interesuje nas skorzystanie z możliwości udostępniania usług znajdujących się na innych urządzeniach. W tym celu należy utworzyć “site”, czyli połaczenie z danym urządzeniem. Na stronie głównej dashboardu Pangolina, należy kliknąć przycisk “Create Site”, po czym nadać mu nazwę oraz potencjalnie zmienić automatycznie wygenerowany zakres adresów IP wykorzystywanych przez utworzoną “pod spodem” sieć WireGuard, jak zostało to pokazane na rysunku 9. Poniżej znajdują się również losowo wygenerowane dane wykorzystywane w celu połączenia się do serwera.



Rysunek 9: Tworzenie ”site” w Pangolinie

Dane te należy zapisać w miejscu docelowym - w naszym przypadku będzie to klaster Kubernetes. Pangolin ma przygotowany poradnik do uruchomienia klienta o nazwie Newt na klastrze Kubernetes, dostępny pod adresem: <https://docs.pangolin.net/manage/sites/install-kubernetes>. W tym celu tworzymy plik ‘newt-cred.yaml’, o zawartości widocznej na rysunku 10.

```
newt-cred.env
1 PANGOLIN_ENDPOINT=https://pangolin.google.sex.pl
2 NEWT_ID=abcdefghijklmnopqrstuvwxyz0123456789abcdefghijklmn
3 NEWT_SECRET=abcdefghijklmnopqrstuvwxyz0123456789abcdefghijklmn
4
```

Rysunek 10: Zapisanie credentiali Pangolina w pliku YAML

Dane z tego sekretu są następnie wykorzystane w pliku values.yaml dla Newta (klienta Pangolina), widocznego na rysunku 11, który pozwala na skonfigurowanie Newta do połączenia się z serwerem Pangolina.

```

values-newt.yaml
1   newtInstances:
2     - name: main
3       enabled: true
4       auth:
5         existingSecretName: newt-cred
6         keys:
7           endpointKey: PANGOLIN_ENDPOINT
8           idKey: NEWT_ID
9           secretKey: NEWT_SECRET

```

Rysunek 11: Plik values.yaml dla Newt'a (klienta Pangolina)

Z przygotowanymi plikami konfiguracyjnymi, możliwe jest ich zaaplikowanie poprzez użycie komend widocznych na rysunku 12. Kluczowe jest uprzednie dodanie repozytorium Pangolina do Helma, co pozwala na zainstalowanie Newt'a przy pomocy jednej komendy Helm.

```

6 # dodanie repo
7
8 helm repo add fossorial https://charts.fossorial.io
9 helm repo update
10
11 # postwaienie newt
12
13 kubectl create secret generic newt-cred -n newt --from-env-file=newt-cred.env
14
15 helm install my-newt fossorial/newt \
16   -n newt --create-namespace \
17   -f values-newt.yaml

```

Rysunek 12: Komendy do uruchomienia Newt'a (klienta Pangolina) na klastrze Kubernetes

Po chwili, w dashboardzie Pangolina powinien pojawić się nowy "site", widoczny na rysunku 13.

Name	Online	Site	Data In	Data Out	Connection Type	Exit Node	Address
Local	● Online	lorem-ipsum-dolor	21.37 GB	21.37 GB	Newt	Exit Node /f90+o2	1.2.3.4
Local-No-Newt	-	sit-amet-consectetur	-	-	Local	-	5.6.7.8
K8s	● Online	adipiscing-ellit-donec	21.37 GB	21.37 GB	Newt v1.5.0	Exit Node /f90+o2	9.10.11.12
TrueNAS	● Online	blandit-turpis-nulla	21.37 GB	21.37 GB	Newt v1.5.2	Exit Node /f90+o2	13.14.15.16

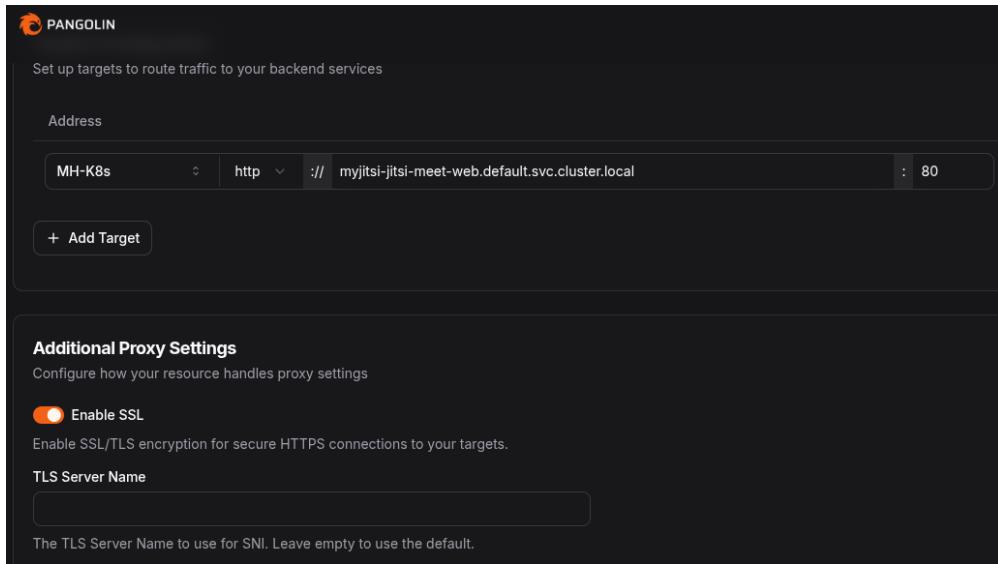
Rysunek 13: Utworzony site (K8s) widoczny w dashboardzie Pangolina

Następnie możliwe jest utworzenie dostępu do strony Jitsi poprzez Pangolina. W tym celu należy przejść do zakładki “Resources” w dashboardzie Pangolina, a następnie kliknąć przycisk “Create Resource”. Pojawi się formularz, widoczny na rysunku 14, w którym należy podać nazwę zasobu, typ (w naszym przypadku HTTP), domenę pod którą ma być dostępne Jitsi (w naszym przypadku jitsi.google.sex.pl), oraz adres pod który będą przekierowane żądania (w naszym przypadku adres usług Jitsi w klastrze Kubernetes, czyli myjitsi-jitsi-meet-web.default.svc.cluster.local). Wykorzystujemy w tym przypadku wewnętrzny adres usługi Kubernetes, ponieważ Newt (klient Pangolina) działa na klastrze i ma dostęp do jego sieci oraz jest w stanie rozwiązać ten adres dzięki wewnętrzniemu DNS Kubernetes.

The screenshot shows the 'Create Resource' interface in Pangolin. The 'Resource Information' section has 'Name' set to 'jitsi'. In the 'Resource Type' section, 'HTTPS Resource' is selected. The 'HTTPS Settings' section shows 'Subdomain' as 'jitsi' and 'Base Domain' as 'google.sex.pl'. The 'Targets Configuration' section lists a target with 'Address' as 'TrueNAS' and 'Port' as ':80', with 'Health Check' set to 'Unknown' and 'Enabled' checked.

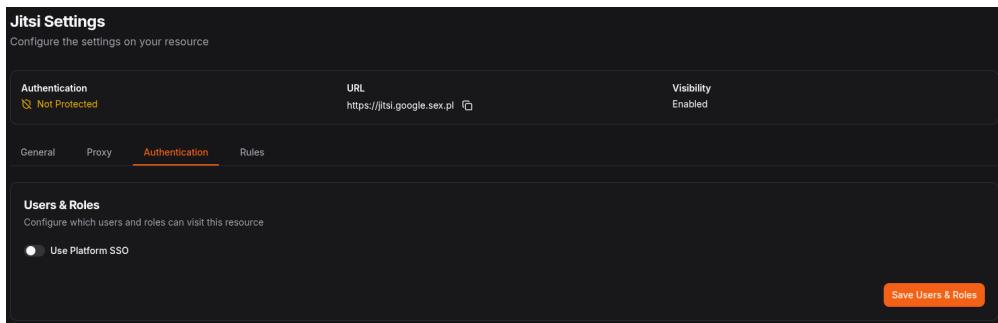
Rysunek 14: Tworzenie dostępu do Jitsi poprzez Pangolina

Utworzony zasób ma automatycznie włączone TLS, widoczne na rysunku 15, dzięki czemu dostęp do Jitsi będzie możliwy poprzez HTTPS.



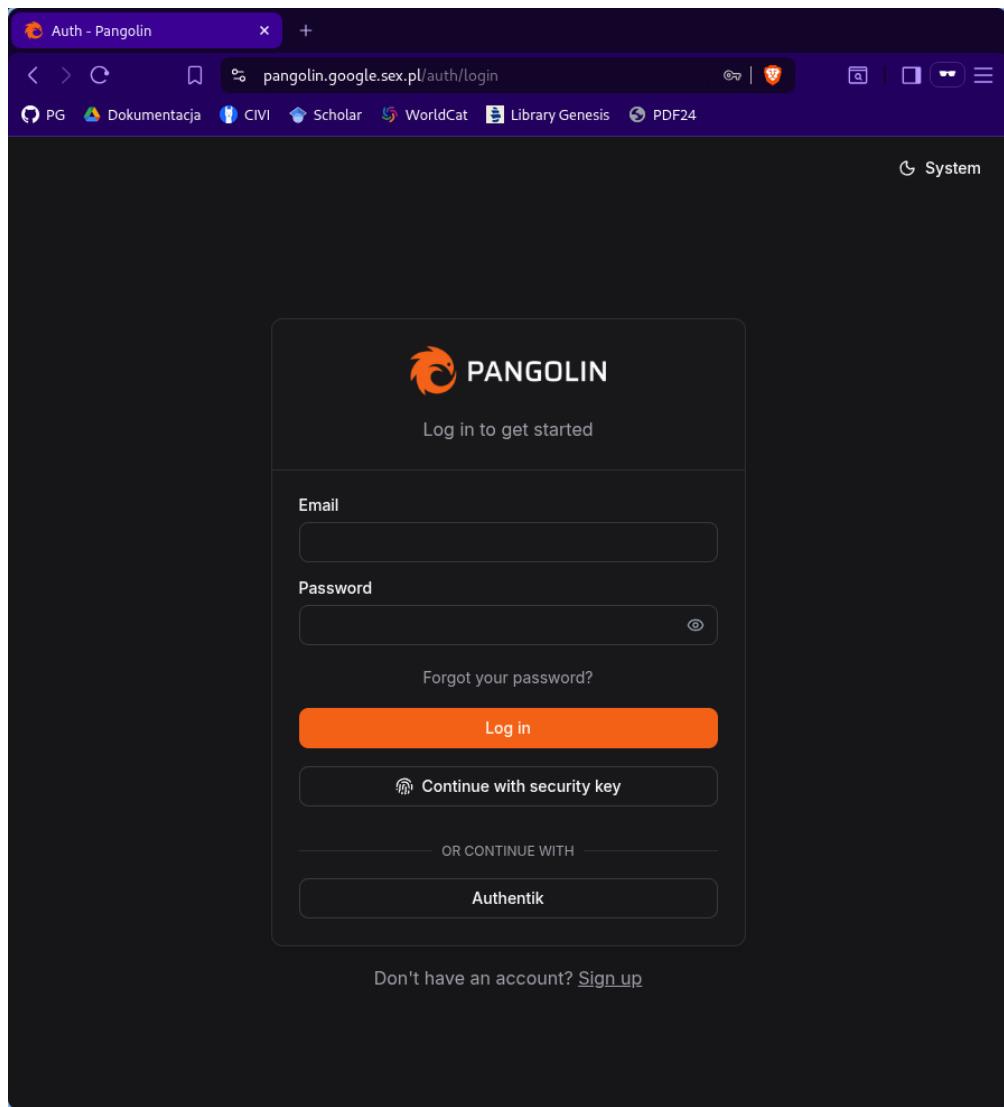
Rysunek 15: Konfiguracja TLS dla dostępu do Jitsi poprzez Pangolina

Ze względu na stosowanie w celu publicznego wystawiania aplikacji oraz jednoczesnego kontroli dostępu do nich, Pangolin domyślnie włącza SSO (Single Sign-On) dla wszystkich udostępnianych aplikacji. W naszym przypadku nie jest to potrzebne, więc należy tę opcję wyłączyć, co zostało pokazane na rysunku 16.



Rysunek 16: Wyłączenie SSO dla dostępu do Jitsi poprzez Pangolina

W przypadku pozostawienia domyślnej opcji włączenia SSO, przy próbie dostępu do Jitsi pojawi się okno logowania, widoczne na rysunku 17.



Rysunek 17: Wygląd okna SSO przy włączonej opcji dla dostępu do Jitsi poprzez Pangolina

Po takiej konfiguracji, możliwe jest wejście na stronę naszego Jitsi oraz utworzenie pokoju. Rozmowę można jednak prowadzić jedynie przez około minutę, po czym użytkownik jest rozłączany z powodu braku dostępu do JVB. Dzieje się tak, ponieważ nie zostało skonfigurowane przekierowanie ruchu multimedialnego do JVB w klastrze Kubernetes.

Aby rozwiązać ten problem, skorzystamy z Tailscale.

Tailscale wymaga utworzenia konta, po czym możliwe jest wygenerowanie klucza sekretu, który pozwoli na automatyczne dołączenie urządzenia do sieci Tailscale. Na urządzeniu które chcemy dołączyć do sieci Tailscale należy zainstalować Tailscale, a następnie uruchomić go z wygenerowanym sekretem.

```
curl -fsSL https://tailscale.com/install.sh | sh  
sudo tailscale up --authkey <TAILSCALE_AUTH_KEY>
```

Listing 3: Komendy do instalacji i uruchomienia Tailscale

Te komendy zostały użyte na serwerze z publicznym IP.

Tailscale oferuje zestaw możliwości uruchomienia na klastrze Kubernetes, opisany dokładnie pod adresem: <https://tailscale.com/learn/managing-access-to-kubernetes-with-tailscale>. Z tego poardnika skorzystamy z opcji uruchomienia Tailscale jako proxy do konkretnej usługi Kubernetes (nie całej sieci), minimalizując w ten sposób potencjalne zagrożenia bezpieczeństwa. Aby połączyć Tailscale z JVB w klastrze Kubernetes, należy utworzyć sekret Kubernetes, który będzie przechowywał klucz autoryzacyjny Tailscale (inny niż użyty na serwerze publicznym), widoczny na rysunku 18.

```
tailscale-secret.yaml  
1  apiVersion: v1  
2  kind: Secret  
3  metadata:  
4    name: tailscale-auth  
5  stringData:  
6    TS_AUTHKEY: tskey-auth-abcdefghijklmnopqrstuvwxyz1234567890abcdefghijklmn
```

Rysunek 18: Sekret Tailscale dla JVB

Dodatkowo rekomendowane jest stworzenie odpowiednich ról RBAC, które pozwolą Tailscale na aktualizację sekretu z adresem IP przydzielonym przez Tailscale, widocznych na rysunku 19.

```
📄 tailscale-rbac.yaml
1   apiVersion: v1
2   kind: ServiceAccount
3   metadata:
4     name: tailscale
5
6   —
7
8   apiVersion: rbac.authorization.k8s.io/v1
9   kind: Role
10  metadata:
11    name: tailscale
12  rules:
13    - apiGroups: []
14      resourceNames: ["tailscale-auth"]
15      resources: ["secrets"]
16      verbs: ["get", "update", "patch"]
17
18   —
19
20  apiVersion: rbac.authorization.k8s.io/v1
21  kind: RoleBinding
22  metadata:
23    name: tailscale
24  subjects:
25    - kind: ServiceAccount
26      name: tailscale
27  roleRef:
28    kind: Role
29    name: tailscale
30    apiGroup: rbac.authorization.k8s.io
```

Rysunek 19: Konfiguracja RBAC dla Tailscale w celu aktualizacji sekretu

Następnie można skonfigurować uruchomienie samego Tailscale, na bazie rekomendowanych ustawień z poradnika powyżej, widocznego na rysunku 20 Kluczowe w konfiguracji jest użycie odpowiedniego serviceAccountName, aby pod uruchamiał się pod tymże kontem, kontener inicjalizujący, który zezwala na przekierowanie pakietów IP (co jest wymagane dla działania Tailscale w trybie proxy) oraz zmienne środowiskowe - klucz autentykujący oraz adres IP usługi, do której chcemy przekierować ruch.

```

tailscale-proxy.yaml
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: tailscale-proxy
5  spec:
6    serviceAccountName: tailscale
7    initContainers:
8      - name: sysctler
9        image: busybox:latest
10       securityContext:
11         privileged: true
12         command: ["/bin/sh"]
13       args:
14         - -c
15         - sysctl -w net.ipv4.ip_forward=1 net.ipv6.conf.all.forwarding=1
16    containers:
17      - name: tailscale
18        image: ghcr.io/tailscale/tailscale:latest
19        env:
20          - name: TS_KUBE_SECRET
21            value: tailscale-auth
22          - name: TS_AUTHKEY
23            valueFrom:
24              secretKeyRef:
25                name: tailscale-auth
26                key: TS_AUTHKEY
27          - name: TS_USERSPACE
28            value: "false"
29          - name: TS_DEST_IP
30            value: 10.108.40.240
31        securityContext:
32          privileged: true
33          capabilities:
34            add:
35              - NET_ADMIN

```

Rysunek 20: Plik konfiguracyjny Tailscale Proxy do przekierowania ruchu do JVB

Znalezienie usługi możliwe jest poprzez komendę kubectl get services, gdzie wyświetlony będzie również jej adres IP. Na rysunku 21 pokazany jest wynik działania tejże komendy, wraz z zaznaczonym adresem IP JVB, który został użyty w celu uruchomienia proxy Tailscale.

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.76.0.1	<none>	443/TCP	26d
myjitsi-jitsi-meet-jvb	ClusterIP	10.108.40.240	<none>	10000/UDP	31h
myjitsi-jitsi-meet-jvb-metrics	ClusterIP	10.108.15.188	<none>	9888/TCP	31h
myjitsi-jitsi-meet-web	ClusterIP	10.106.197.43	<none>	80/TCP	31h
myjitsi-prosody	ClusterIP	10.108.213.244	<none>	5280/TCP, 5281/TCP, 5347/TCP, 5222/TCP, 5269/TCP	31h

Rysunek 21: Usługi uruchomione na klastrze Kubernetes, z zaznaczonym adresem IP JVB

Po przygotowaniu powyższych plików można je zaaplikować przy użyciu kilku komend kubectl apply, widocznych na rysunku 22. W przeciwieństwie do Pangolina czy Jitsi, Tailscale wykorzystuje obraz Dockera, więc nie jest potrzebne użycie Helma do jego instalacji.

```

1 # postawienie tailscale
2
3 kubectl apply -f tailscale-secret.yaml
4 kubectl apply -f tailscale-rbac.yaml
5 kubectl apply -f tailscale-proxy.yaml

```

Rysunek 22: Komendy do uruchomienia Tailscale na klastrze Kubernetes

Po uruchomieniu komend możliwa jest komunikacja między serwerem z publicznym IP oraz klastrem poprzez sieć Tailscale. Odpowiednie urządzenie pojawia się również w panelu administratorskim Tailscale, widoczne na rysunku 23.

MACHINE	ADDRESSES	VERSION	LAST SEEN	...
tailscale-proxy	100.105.219.104	1.90.9 Linux 6.12.48+deb13-amd64	Connected	

Rysunek 23: Urządzenie widoczne w panelu Tailscale po uruchomieniu klienta na klastrze Kubernetes

Ustanowiona komunikacja umożliwia przekierowanie ruchu wchodzącego na port 10000 protokołu UDP serwera z publicznym IP do JVB postawionego w klastrze. W tym celu skorzystamy z iptables, w którym takie przekierowanie możliwe jest przy użyciu dwóch komend, widocznych na rysunku 24. Ze względu na dynamiczny routing jaki umożliwia Tailscale, wymagane jest jedynie podanie adresu IP, takiego jaki pokazał się w panelu administracyjnym na rysunku 23.

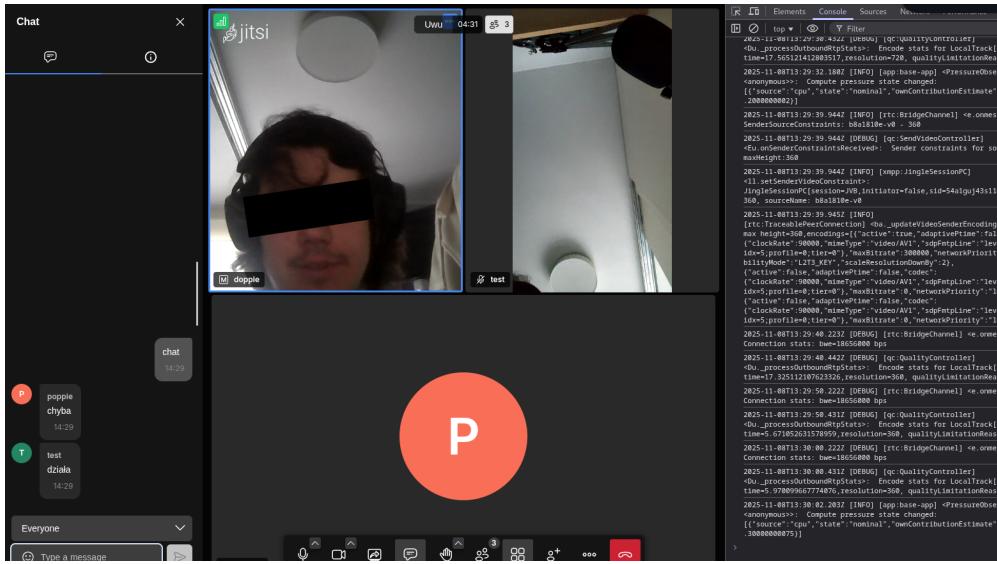
```

30 # routing w iptables
31
32 iptables -t nat -A PREROUTING -p udp --dport 10000 -j DNAT --to-destination 100.105.219.104:10000
33 iptables -t nat -A POSTROUTING -j MASQUERADE
```

```

Rysunek 24: Komendy iptables do przekierowania ruchu z serwera publicznego do JVB w klastrze Kubernetes

W ten sposób skonfigurowane proxy na serwerze umożliwia komunikację użytkownika z JVB, czego efektem jest możliwe przeprowadzenie rozmowy, której test widoczny jest na rysunku 25. Rozmowa była przeprowadzona przy użyciu dwóch urządzeń w tej samej sieci - laptopa oraz komputera desktop, a także telefonu połączonego do innej sieci przez 5G.



Rysunek 25: Działająca rozmowa w Jitsi, dostępna publicznie

### 3.4.2 Konfiguracja Jitsi z Octo

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

`octo:`

`enabled: True`

Przeskalowanie samego JVB polega na dodaniu liczby instancji w jvb.replicaCount:

`jvb:`

`replicaCount: 2`

Jednym z wymogów Octo jest to, aby każdy JVB ogłaszał stały, routowalny adres IP, pod którym jest osiągalny przez inne JVB oraz klientów. Opcja ta powoduje, że JVB wykorzystuje zewnętrzny adres IP węzła (Node IP), a nie adres IP dla Pod, który nie jest routowalny pomiędzy węzłami ani dla klientów.

`jvb:`

`# Każdy Pod JVB ogłasza wyłącznie zewnętrzny adres IP swojego Node  
  useNodeIP: true`

Dodatkowo JVB musi wystawiać swój port bezpośrednio na węźle, na którym jest uruchomiony. Użycie hostPort sprawia, że port UDP Jitsi Videobridge jest dostępny bezpośrednio na adresie IP węzła. Jednocześnie oznacza to, że na jednym węźle może działać tylko jedna instancja JVB, ponieważ ten sam port nie może być współdzielony przez wiele podów.

`jvb:`

`# Wystaw port interfejsu JVB na świat zewnętrzny tylko na węzłach,  
  # które faktycznie go posiadają  
  useHostPort: true`

## **3.5 Monitoring**

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

### **3.5.1 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 pojedynczych komponentów), a także zapewnia odporność na przerwę w pracy, ponieważ pozwala na niedostępność pojedynczych 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 4: 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

### 3.5.2 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 5: 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 6: 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
```

### 3.5.3 Aplikacja prezentowania danych

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

```

```

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

Listing 7: 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.

Użyliśmy istniejącego już panelu (ang. dashboard) Grafany dla Jitsi: <https://grafana.com/grafana/dashboards/12098-jitsi-meet/>, przedstawiającego zebrane metryki z Jitsi. Poniżej zaprezentowano jak zaimportować panel w programie Grafana:

Należy pobrać plik JSON ze strony podanej powyżej.

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

The screenshot shows the Grafana interface with the title bar 'Grafana'. On the left is a sidebar with links: Home, Bookmarks (with a note 'Bookmark pages for them to appear here'), Starred, Dashboards (which is selected and highlighted with an orange border), and Explore. The main area is titled 'Dashboards' with the sub-instruction 'Create and manage dashboards to visualize your data'. It includes a search bar, a filter by tag dropdown, a 'Starred' checkbox, and buttons for 'New', 'New dashboard', 'New folder', and 'Import'. The 'Import' button is highlighted with a dark gray background.

Po przejściu do opcji “Import” wystarczy wysłać plik JSON panelu oraz wcisnąć “Load”.

The screenshot shows the 'Import dashboard' dialog within the Grafana interface. The title bar says 'Home > Dashboards > Import dashboard'. The left sidebar is identical to the previous screenshot. The main area has a title 'Import dashboard' and a sub-instruction 'Import dashboard from file or Grafana.com'. It features a dashed box for 'Upload dashboard JSON file' with a note 'Drag and drop here or click to browse Accepted file types: .json, .txt'. Below it is a link 'Find and import dashboards for common applications at [grafana.com/dashboards](https://grafana.com/dashboards)' and a field 'Grafana.com dashboard URL or ID' with a 'Load' button next to it. At the bottom is a section 'Import via dashboard JSON model' containing a JSON code snippet. At the very bottom are 'Load' and 'Cancel' buttons, with 'Load' being highlighted.

## **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 8: Skrypt do testowania Jitsi  
29

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:

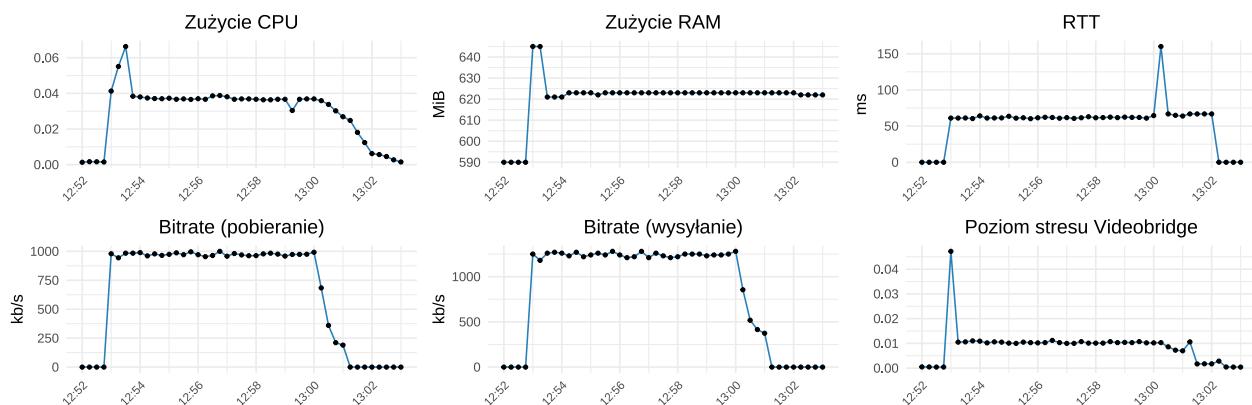
1. 1 konferencja 3 uczestników - testowa konferencja,
2. 1 konferencja 10 uczestników - przy porównaniu z 1 konferencją i trzema uczestnikami uzyskamy sens wielkości w jakich się poruszamy,
3. 3 konferencje po 10 uczestników - sprawdzenie możliwości, w przypadku optymalizacji przetestowanie gdzie zostaną 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:

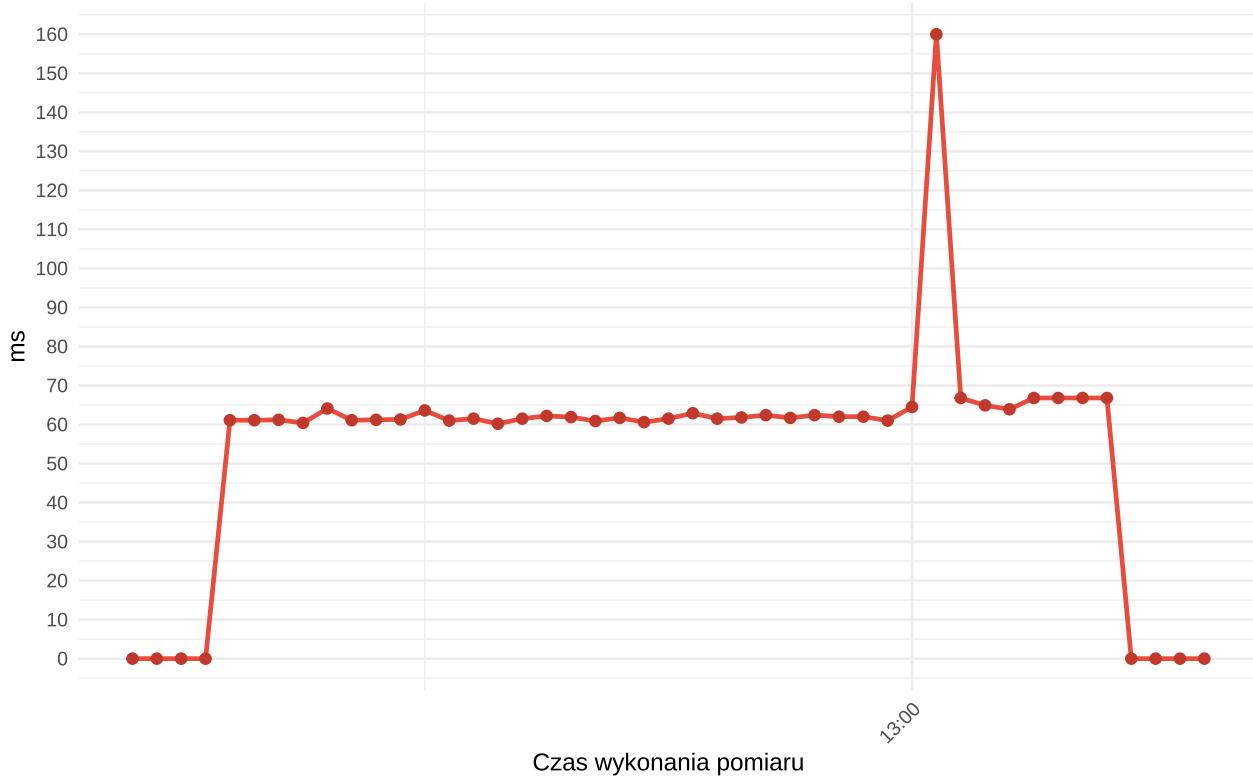
- zużycie CPU - w Kubernetes wartość CPU nie jest podawana w procentach całego serwera, lecz w jednostkach rdzeni logicznych (vCPU lub Core),
- poziom stresu Videobridge - całkowity obraz jak wykorzystywany jest Videobridge, gdzie 0 oznacza brak obciążenia, a 1 pełne wykorzystanie zasobów (chociaż wartości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

## 4.2 Wyniki oraz spostrzeżenia sprzed optymalizacji za pomocą Octo



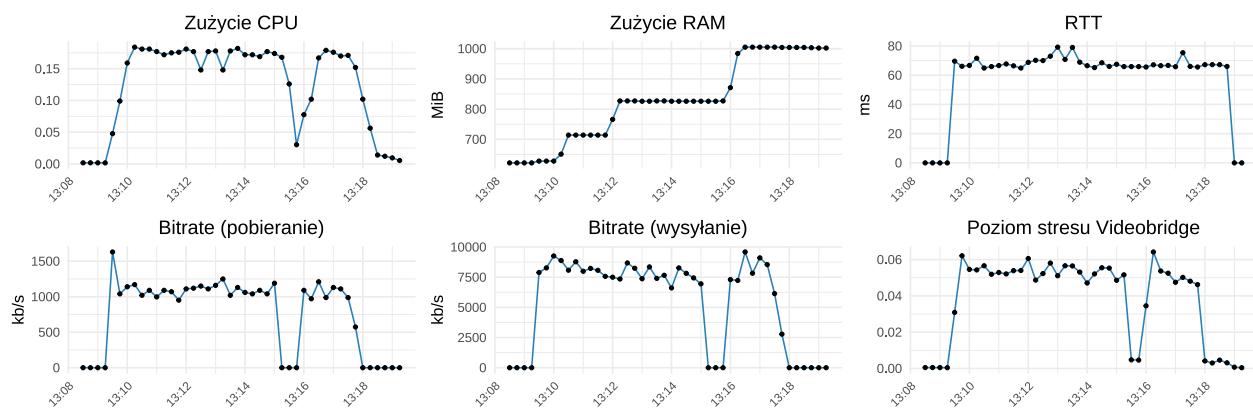
Rysunek 26: Scenariusz 1 - 1 konferencja, 3 uczestników

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 27: 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.

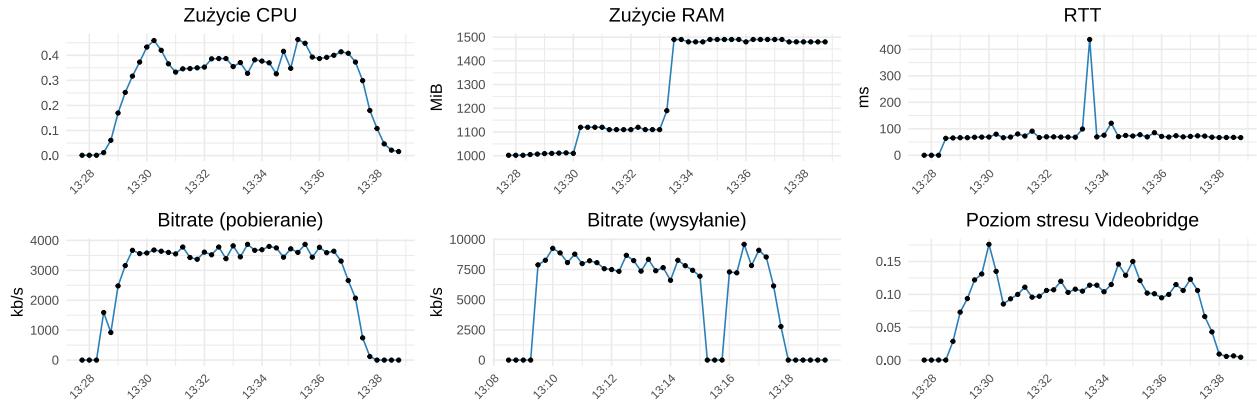


Rysunek 28: Scenariusz 2 - 1 konferencja, 10 uczestników

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.

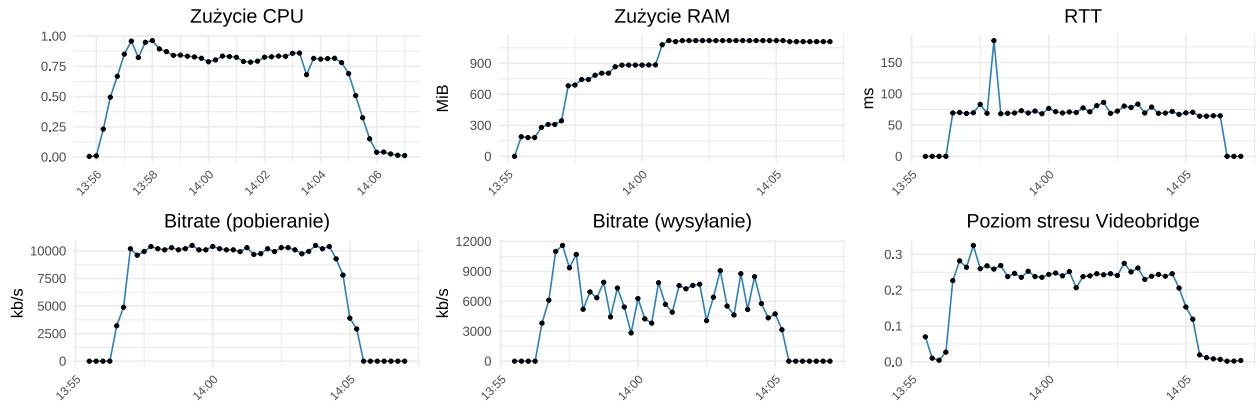
Warto zauważyc, że zużycie RAMu jest wysokie i nie spada - taka sytuacja dzieje się także na następnym pomiarze.



Rysunek 29: 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.

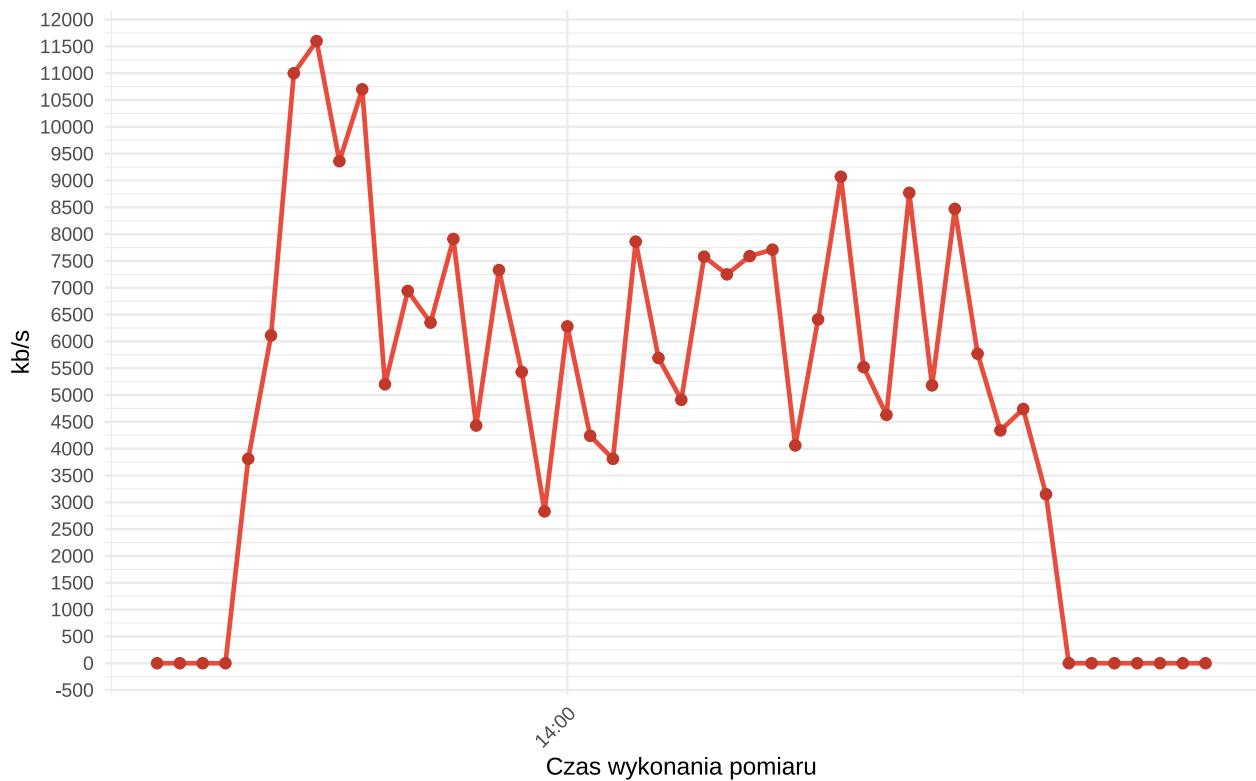
Przy tych wykresach warto zapamiętać zużycie CPU na poziomie 0.4, poziom stresu na poziomie 0.1 oraz bitrate na poziomie 4000 (pobieranie), przydadzą się do następnych wniosków.



Rysunek 30: Scenariusz 4 - 10 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 31: Bitrate (wysyłanie) dla scenariusza 4

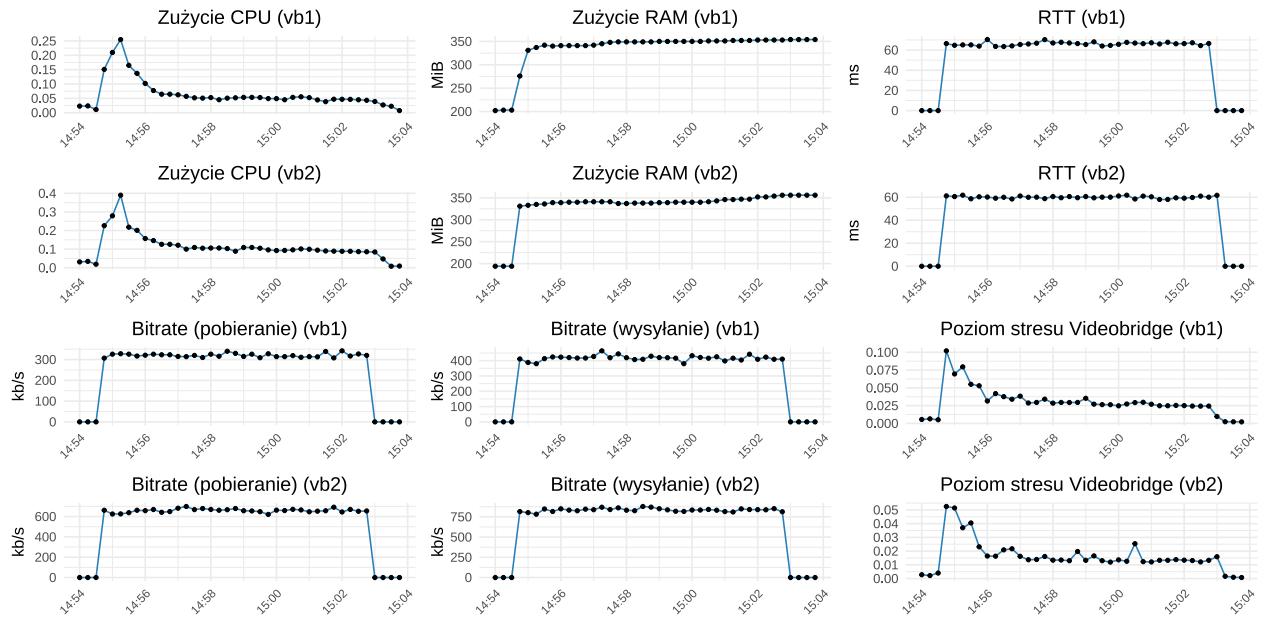
Ciężko tutaj o jednoznaczną przyczynę, lecz teoria nad tym zostanie omówiona w rozdziale “Wnioski”.

### 4.3 Wyniki oraz spostrzeżenia po optymalizacji za pomocą Octo

Dopiski w nawiasie vbX oznaczają dane z którego z Videobridge są prezentowane, gdzie X to number Videobridge.

W tej sytuacji posiadamy instancję Jitsi z jej funkcją “Octo”, wewnętrznie przez twórców nazywaną “Relay”, która umożliwia połączenie wielu Videobridge tak, aby mogły rozdzielić między siebie całość ruchu.

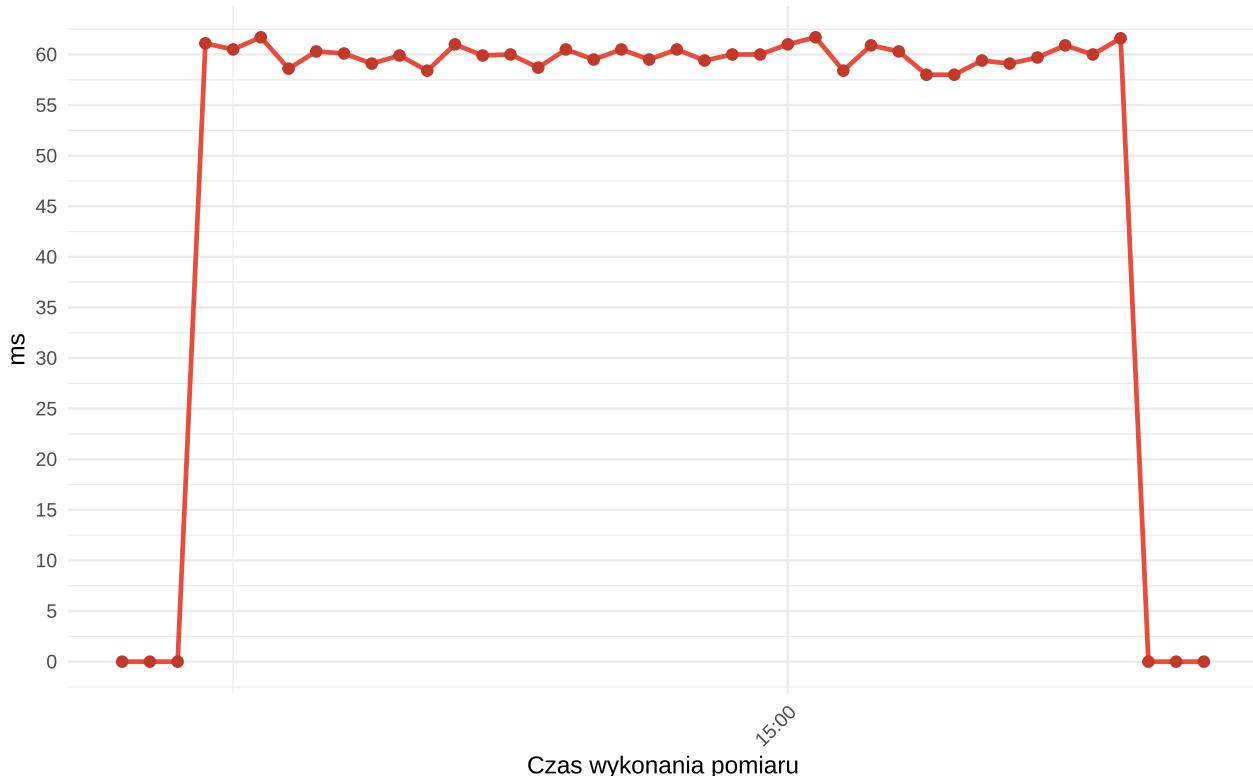
Niefortunnie, nie byliśmy w stanie otrzymać scenariusza w którym na jednym Videobridge istnieje jedna konferencja. Udało się nam to w testowaniu ręcznym przed zrobieniem dokumentu, lecz mogło to mieć związek z błędną konfiguracją (podany adres bezpośrednio do jednego z Videobridge, zamiast ogólnie do serwisu który mógł rozłożyć ruch).



Rysunek 32: Scenariusz 1 - 1 konferencja, 3 uczestników

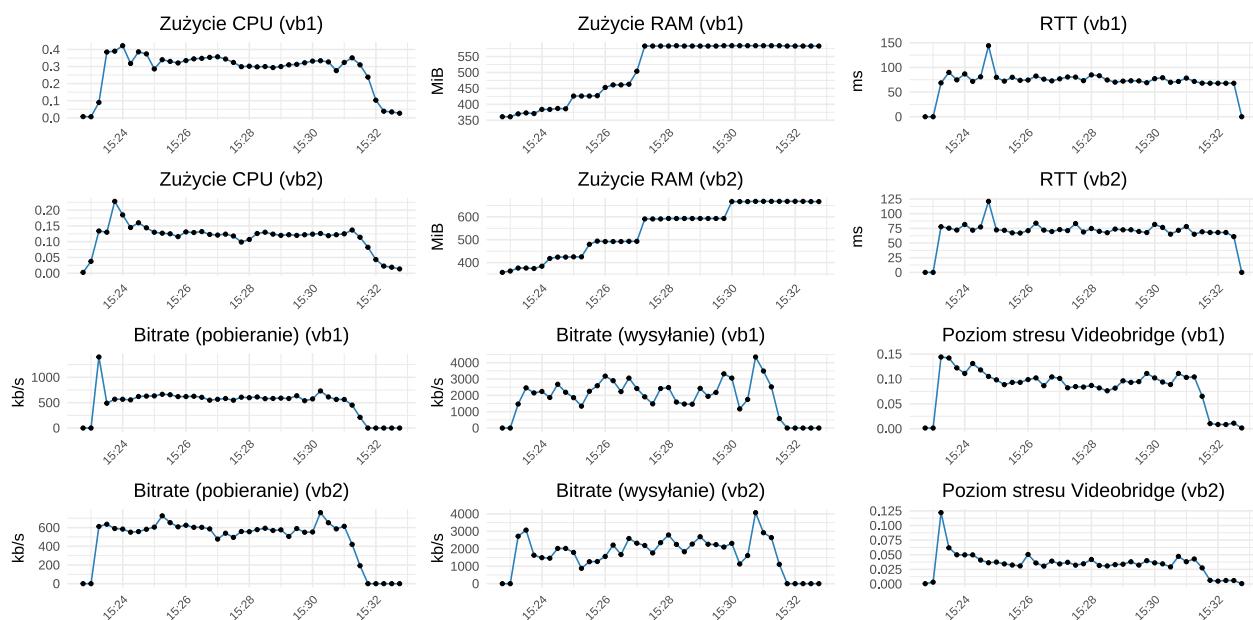
Już na starcie widać korelację między obydwojoma instancjami. Zużycie wzrasta na obydwiu równomiernie, lecz vb1 konsekwentnie jest o połowę mniejszej obciążone niż vb2. Jak poprzednio, korelacja z poziomem stresu jest taka sama. Przyjrzyjmy się RTT dla obydwu sytuacji.





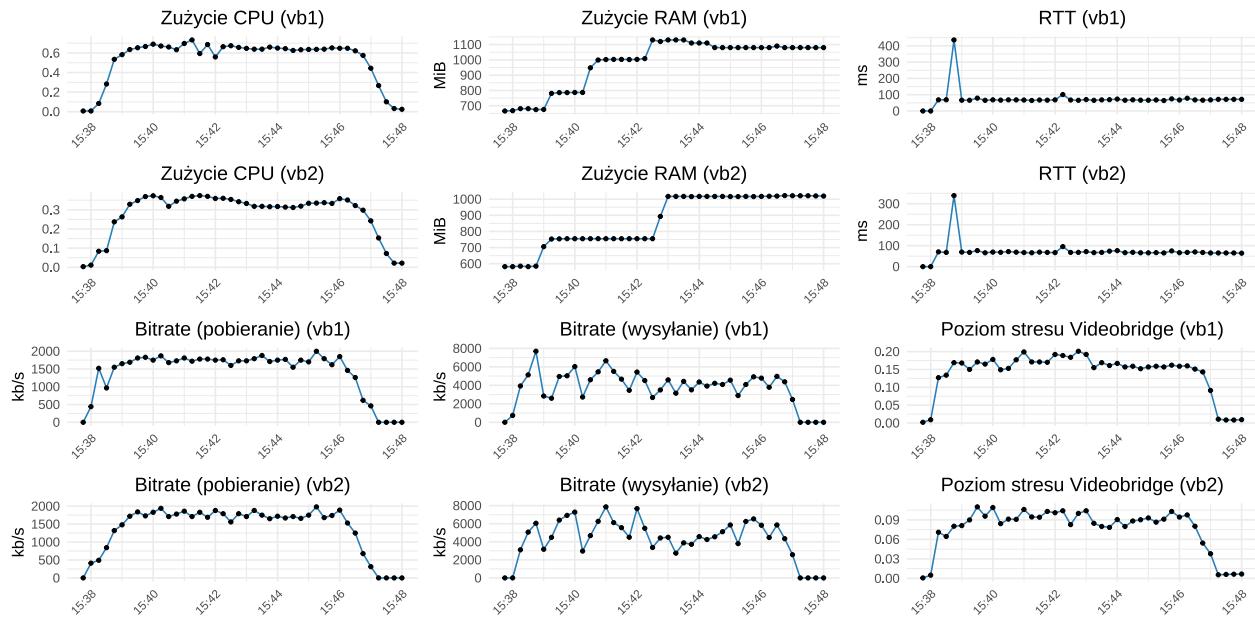
Rysunek 34: Scenariusz 1 OCTO - RTT (vb2)

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



Rysunek 35: 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 36: Scenariusz 4 - 3 konferencje po 10 uczestników

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

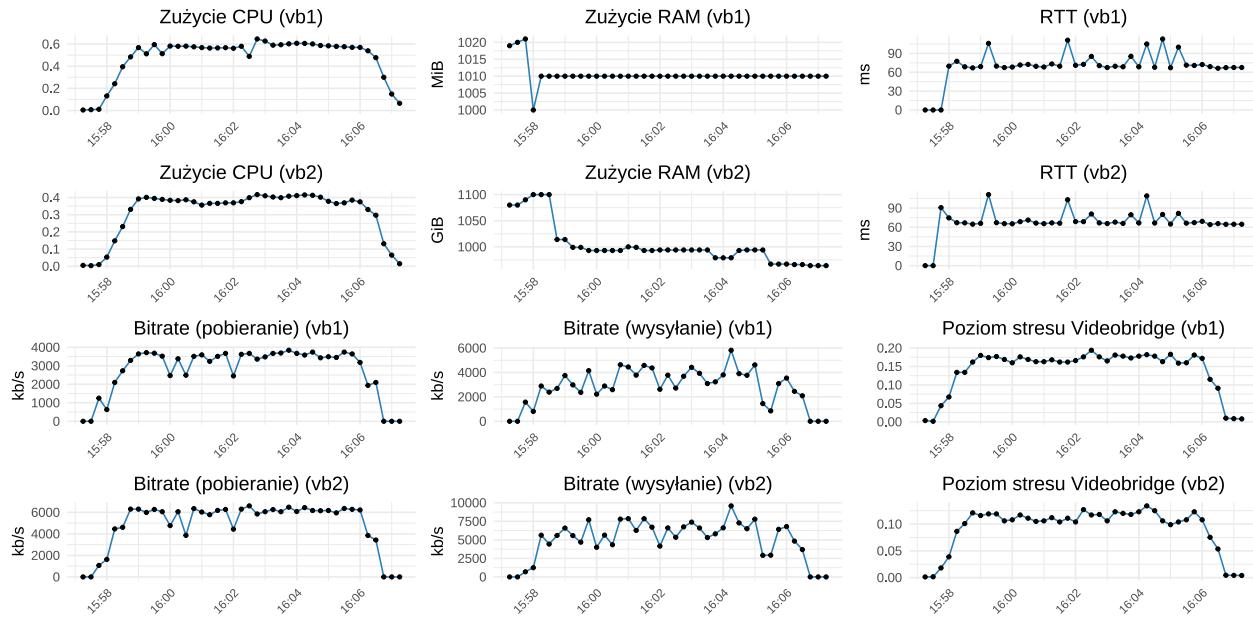


Rysunek 37: Scenariusz 1 OCTO - RTT (vb1)



Rysunek 38: Scenariusz 1 OCTO - RTT (vb2)

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



Rysunek 39: Scenariusz 4 - 10 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 31 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.