Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мережних технологій факультету інформатики

СИСТЕМА МОНІТОРИНГУ МЕРЕЖІ ПІДПРИЄМСТВА

Текстова частина до кваліфікаційної роботи

за спеціальністю „Комп'ютерні науки" 122

Керівник кваліфікаційної роботи

к.т.н., доц. Черкасов Д.І.

(підпис)

"____"_____2023 р.

Виконала студентка КН-4

Загоруйко О.О.

"____"_____2023 р.

Київ 2023

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мережних технологій факультету інформатики

ЗАТВЕРДЖУЮ

Зав. кафедри мережних технологій,

проф., доктор фіз.-мат. наук

_____Малашонок Г. І.

(підпис)

„____”_____2023 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на кваліфікаційну роботу

студентки 4 року навчання БП "Комп’ютерні науки"

Загоруйко Ольги Олегівни

на тему:

Система моніторингу мережі підприємства

Зміст ТЧ до кваліфікаційної роботи:

Індивідуальне завдання
Анотація
Вступ
1. Порівняльний аналіз систем моніторингу
2. Структурна розробка власного рішення для моніторингу
3. Створення програмного компоненту системи
Висновки
Список використаних джерел
Додатки (за необхідністю)

Дата видачі: „___” _____ 2022 р.

Керівник: к. т. н. Черкасов Д.І._____ (підпис)

Завдання отримала _____ (підпис)

Календарний план

№	Етап кваліфікаційної роботи	Термін виконання етапу
1.	Отримання теми кваліфікаційної роботи	20.12.2022
2.	Огляд наявних систем моніторингу	12.01.2023
3.	Аналіз особливостей наявних систем	31.01.2023
4.	Написання вступу та першого розділу	09.02.2023
5.	Аналіз технічних рішень	23.02.2023
6.	Розробка структурної схеми власного рішення для моніторингу	07.03.2023
7.	Написання другого розділу	25.03.2023
8.	Розробка компонента системи	14.04.2023
9.	Написання третього розділу та висновків	08.05.2023
10.	Створення презентації	19.05.2023
11.	Подання роботи на кафедру для перевірки на плагіат	22.05.2023
12.	Захист кваліфікаційної роботи	29.05.2023

Студент _____

Керівник _____

“_____” _____

Зміст

Анотація **5**

Вступ **7**

РОЗДІЛ 1. Порівняльний аналіз систем моніторингу **10**

1.1.Система моніторингу Zabbix 10

1.2.Система моніторингу Nagios 12

1.3.Система моніторингу Prometheus 14

1.4. Порівняльний аналіз 16

РОЗДІЛ 2. Структурна розробка власного рішення для моніторингу **19**

2.1. Розподілений моніторинг у Zabbix 19

2.2. Висока доступність 22

2.3. Підходи до управління базами даних 24

2.4. Кластеризація бази даних 26

2.5. Galera Cluster 29

2.6. Структурна схема 33

РОЗДІЛ 3. Створення програмного компоненту системи **37**

3.1. Розгортання системи моніторингу 37

3.2. Створення застосунку автентифікації користувачів 40

3.3. Імітація трафіку 43

3.4. Реалізація моніторингу 43

3.5. Результати 44

Висновки **47**

Список використаних джерел **49**

Додатки **50**

Анотація

Дана кваліфікаційна робота присвячена галузі мережних технологій. У ній на структурному рівні розроблено систему моніторингу мережі підприємства, яка складається з головного кампусу і територіально рознесених офісів. Система моніторингу має високу доступність за рахунок стійкості до відмов окремих компонентів.

У головному офісі створюється відмовостійка конфігурація з двох серверів, які функціонують по схемі active-standby, та базою даних у вигляді кластеру. Для відмовостійкості система також містить 2 фронтенда. Доступ до яких здійснюється через балансувальники навантаження. Моніторинг в регіональних офісах здійснюється за допомогою проксі.

У рамках дослідження виконано порівняння існуючих прототипів до системи. Після якого було обрано взяти за основу компоненти Zabbix як бази для побудови власного рішення. Та сформувано вимоги щодо функціональності системи: можливість збору метрик, наявність веб-інтерфейсу, надсилання сповіщень, середня складність, перевірка станів, підтримка моніторингу розподілених мереж.

Проведено детальний аналіз технічних рішень, які дозволяють реалізувати поставлені вимоги.

Розроблено тестову конфігурацію системи щоб продемонструвати роботу моніторингу. Для цього за допомогою технології контейнеризації Docker було розгорнуто наступні сервіси: MariaDB Galera Cluster, Zabbix Server, Zabbix Web та агент Zabbix. Агент Zabbix збирає дані моніторингу та відправляє їх на сервер. Сервер Zabbix отримує, обробляє та зберігає ці дані в MariaDB, а Zabbix Web дозволяє переглядати зібрані дані та керувати ними. Таким чином було продемонстровано можливості моніторингу за допомогою агента.

Щоб продемонструвати можливості безагентного моніторингу, за допомогою мови програмування Python створено застосунок для автентифікаціїї

користувачів з використанням фреймворку FastApi. У ньому є три кінцеві точки для реєстрації, входу та виходу користувача. Дані про користувачів зберігаються у базі даних MySQL. До цього застосунку, був написаний сценарій, який імітує трафік для функцій login/logout, надсилаючи HTTP-запити на локальний сервер. Також було створено Python сценарій для паралельного моніторингу та збору показників про даний трафік, який надсилає зібрані метрики до Zabbix з використанням модуля pyzabbix.

Вступ

Цифровий світ побудований на комунікаційних мережах, які дозволяють тисячам людей та організацій підключатися та взаємодіяти по всьому світу. Вирішальним фактором успішності організації на сучасному конкурентному ринку, як правило, є задоволення потреб клієнтів та постачальників. Тому, щоб покращити взаємодію між працівниками, клієнтами та партнерами, та підтримати критично важливих операцій, організації значною мірою покладаються на свої комунікаційні мережі. Так як особливістю більшості нинішніх підприємств є розподілена структура, тобто необхідно щоб у членів команд був доступ до спільних даних та необхідних корпоративних сервісів з будь-якої платформи: комп'ютера, смартфона, планшета. Також, на використанні комунікаційної мережі базується корпоративний документообіг. Розвиток бізнесу та досягнення його довгострокових цілей забезпечується інформаційною інфраструктурою, яка базується на комунікаційній мережі.

Таким чином, комунікаційна мережа є життєво важливим елементом інформаційної інфраструктури сучасного підприємства. Однак, працездатності мережі можуть завадити різномані збої. До факторів, які можуть завадити працездатності мережі відноситься: зловмисницькі дії, недбалість працівників, вихід з ладу обладнання, помилки з програмним забезпеченням та інші. І будь-який із них може суттєво вплинути на роботу організації та призвести до негативних наслідків, таких як простій системи, втрата даних, зниження продуктивності, фінансові втрати, репутаційні збитки та юридична відповідальність.

Щоб захистити організацію, зменшити ці та інші загрози, важливо постійно спостерігати за мережею та виявляти будь-які загрози на ранній стадії, до того, як вони спричинять серйозні проблеми. Впоратися із цим завданням допомагає моніторинг мережі підприємства – процес збору, аналізу та візуалізації даних про стан мережі та всіх підключених до неї пристроїв. Метою моніторингу є

забезпечення безперебійної та ефективної роботи мережі, шляхом виявлення проблем та збоїв, аналізу навантаження на пристрої та канали зв’язку, забезпечення високого рівня безпеки та захисту даних, відстеження та аналізу статистики використання ресурсів мережі, планування майбутніх розширень та обслуговування.

Моніторинг мережі передбачає використання спеціальних інструментів і методів для збору та аналізу даних про мережевий трафік, продуктивність пристроїв та події безпеки. Завдяки цим даним адміністратори визначають вузькі місця, загрози безпеці та інші проблеми, які можуть порушити роботу мережі та призвести до простою.

Ефективне використання мережевих ресурсів і планування майбутніх розширень мережі вимагають постійного моніторингу та аналізу мережевого трафіку, використання пристроїв та інших показників. Відстежуючи статистику використання мережі, спеціалісти можуть виявити недостатньо використані ресурси та спланувати перерозподіл ресурсів у тих областях, де вони потрібні. Це може допомогти уникнути перевантаження мережі та вузьких місць, забезпечуючи оптимальну продуктивність.

Дані, зібрані інструментами моніторингу мережі, зазвичай відображаються на інформаційних панелях або у звітах, що надає адміністраторам можливість переглядати в режимі реального часу стан і продуктивність мережі. Більш того, ці інструменти надсилають сповіщення, при виконанні певних умов, наприклад, коли пристрій переходить у мережу, або коли мережевий трафік перевищує певний поріг.

Слід зазначити, що крім забезпечення моніторингу в реальному часі та сповіщень, деякі інструменти моніторингу мережі можна використовувати для аналізу історії даних мережі. Це може бути корисним для визначення довгострокових тенденцій, оптимізації продуктивності та планування майбутніх оновлень мережі.

Метою дипломної роботи є обґрунтування важливості та дослідження підходів до побудови систем моніторингу комунікаційної мережі сучасного підприємства, інструментів та типових задач моніторингу. На структурному рівні буде виконана розробка системи моніторингу, особливістю якої є відкритість архітектури та використання новітніх технологій. Один з програмних компонентів системи буде розроблено детально для ілюстрації практичності запропонованих структурних рішень.

РОЗДІЛ 1. Порівняльний аналіз систем моніторингу

1.1.Система моніторингу Zabbix

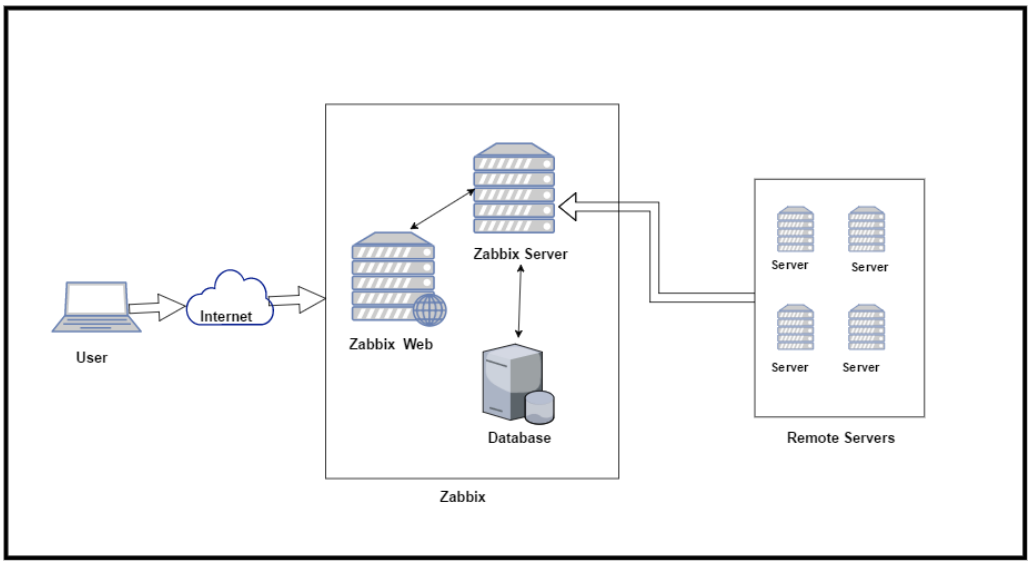
Zabbix — це популярне програмне забезпечення розподіленого моніторингу корпоративного рівня з відкритим вихідним кодом, розроблене для забезпечення моніторингу та аналізу мереж, серверів, програм і служб у реальному часі.

Zabbix пропонує моніторинг тисяч метрик, зібраних з фізичних та віртуальних машин. Більш того, надає централізовану систему моніторингу, яка дозволяє контролювати різні мережеві служби, такі як HTTP, SMTP, POP3 та IMAP, а також сервери, програми та інші пристрої. Він використовує гнучкий механізм сповіщень, щоб попереджати адміністраторів про досягнення порогових значень, а також може автоматично виконувати сценарії або команди для вирішення проблем.

Zabbix використовує клієнт-серверну архітектуру, що складається з кількох компонентів, які працюють разом. Нижче наведено ключові компоненти архітектури Zabbix:

- Сервер Zabbix – центральний компонент архітектури, який збирає та аналізує дані з пристроїв, що контролюються. Він також керує базою даних, зберігає дані моніторингу та створює звіти.
- Zabbix Agent — це невеликий програмний компонент, який працює на контрольованому хості та збирає всю необхідну інформацію локально. Потім надсилаючи її на сервер Zabbix для аналізу.
- Zabbix Proxy — це проміжний компонент, який використовується для моніторингу віддалених серверів. Він може збирати дані про продуктивність і доступність від імені сервера Zabbix та зберігати їх у буфері, а потім надсилати на сервер Zabbix. Таким чином, Zabbix Proxy може взяти на себе частину навантаження та розвантажити сервери Zabbix.

- Веб-інтерфейс — це інтерфейс користувача, який дозволяє користувачам з будь-якого місця та платформи налаштовувати систему моніторингу, переглядати сповіщення, графіки та створювати звіти.
- База даних: сервер Zabbix використовує базу даних для зберігання даних моніторингу, конфігураційної інформації та інших метаданих. Він підтримує кілька баз даних, таких як MySQL, PostgreSQL і Oracle.
- Система сповіщень — це компонент, який надсилає сповіщення користувачам у разі виявлення проблем.



(Рис. 1.1.1 – архітектура Zabbix)[1]

Ключові переваги Zabbix:

- Висока масштабованість, що дозволяє контролювати масштабні корпоративні середовища. Він підтримує розподілений моніторинг, що означає, що кілька серверів Zabbix можна використовувати для моніторингу різних частин інфраструктури, а дані можна агрегувати та переглядати централізовано.
- Zabbix має потужний API та дозволяє створювати власні шаблони моніторингу, які можна застосувати до різних хостів або груп хостів.
- Широкий спектр можливостей моніторингу, включаючи моніторинг на основі агентів, без агентів і на основі SNMP. Він підтримує різні протоколи та технології, такі як ICMP, TCP, UDP, HTTP, FTP, SSH, SMTP та DNS.

- Візуалізація даних доступна у вигляді графіків, екранів, карт та оглядів.

Підсумовуючи, Zabbix є досить потужним та надійним безкоштовним рішенням для моніторингу, яке чудово підходить для корпоративних середовищ. Широкі можливості моніторингу, простота використання, сповіщення в реальному часі, масштабованість і можливості звітування роблять його зручним для організацій, які хотять підтримувати критично важливі системи в робочому стані.

1.2.Система моніторингу Nagios

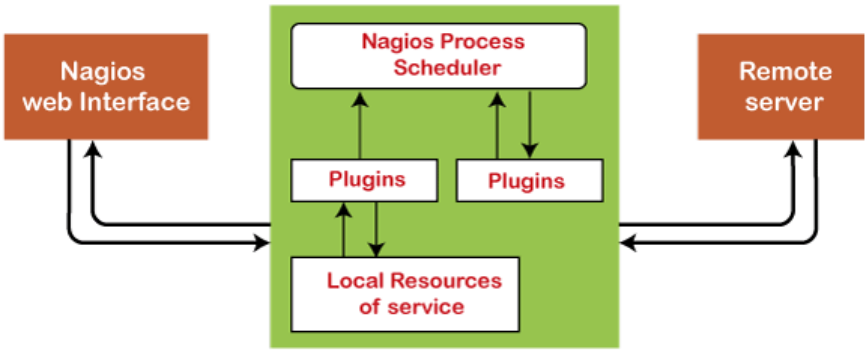
Nagios — це безкоштовне програмне забезпечення з відкритим кодом для постійного моніторингу систем, мереж та інфраструктур. Nagios був створений в 1999 році і з тих пір стало одним з найпопулярніших систем моніторингу. Він запускає плагіни, що зберігаються на сервері, який підключено до хосту або іншого сервера у вашій мережі чи Інтернеті. Nagios відстежує всю вашу IT-інфраструктуру, щоб забезпечити належне функціонування систем, програм, послуг і бізнес-процесів. У разі збою Nagios може попередити технічний персонал про проблему, дозволяючи їм розпочати процеси усунення до того, як збої вплинуть на бізнес-процеси, кінцевих користувачів або клієнтів[2].

Nagios використовує клієнт-серверну архітектуру. Зазвичай сервер Nagios працює на хості, а плагіни працюють на всіх віддалених хостах, які слід контролювати. Плагіни Nagios збирають корисні дані та надсилають їх до планувальника процесів, який відображає інформацію через графічний інтерфейс користувача.

Nagios складається з 3 основних компонентів:

- Планувальник процесів— серверна частина Nagios, яка використовується для перевірки плагінів і надсилання сповіщень відповідно до результату.

- GUI— інтерфейс Nagios, який використовується для відображення веб-сторінок, згенерованих CGI, та керування компонентами IT-інфраструктури.
- Плагіни Nagios поділяються на 2 типи: стандартні та спеціальні плагіни. Стандартні — готові плагіни, які постачаються в комплекті з Nagios і призначені для моніторингу загальних системних компонентів. А користувальницькі плагіни — плагіни, створені користувачами для моніторингу конкретних компонентів або програм, унікальних для їх організації.



(Рис. 1.2.1 – архітектура Nagios)[3]

Деякі з ключових можливостей інструменту моніторингу Nagios:

- комплексний моніторинг програм, служб, операційних систем, мережевих протоколів, системних показників і компонентів інфраструктури;
- ідеальне зберігання даних, оскільки є система керування журналами;
- доступні плагіни графічної обробки даних;
- контролює служби мережі, такі як: SMTP, HTTP, FTP, SSH;
- має додатковий веб-інтерфейс для перегляду сповіщень, файлів журналу тощо;
- відстежує різні ресурси сервера, такі як: пам’ять, процесор , диски та системні журнали;
- забезпечує віддалений моніторинг через зашифровані тунелі SSL і SSH.
- дозволяє відстежувати та усувати проблеми продуктивності сервера;
- допомагає спланувати оновлення інфраструктури до того, як застарілі системи спровокують збої;

- автоматично виправляє проблеми в критичній ситуації;
- може бути запущений на будь-якій операційній системі;
- відстежує різні сервери баз даних, такі як SQL Server, MySQL.[3]

1.3.Система моніторингу Prometheus

Prometheus — це технологія з відкритим вихідним кодом, розроблена для забезпечення функцій моніторингу та оповіщення для хмарних середовищ, зокрема Kubernetes. Він був розроблений у 2012 році командою SoundCloud і зараз підтримується Cloud Native Computing Foundation (CNCF). Даний продукт призначений для збору та моніторингу показників із різних джерел, таких як сервери, бази даних, програми та мережеві пристрої. Він підтримує широкий спектр джерел даних і протоколів, таких як HTTP, HTTPS, DNS, SNMP і JMX. Він може збирати та зберігати величезну кількість часових рядів даних і забезпечувати в режимі реального часу видимість продуктивності складного IT-середовища.

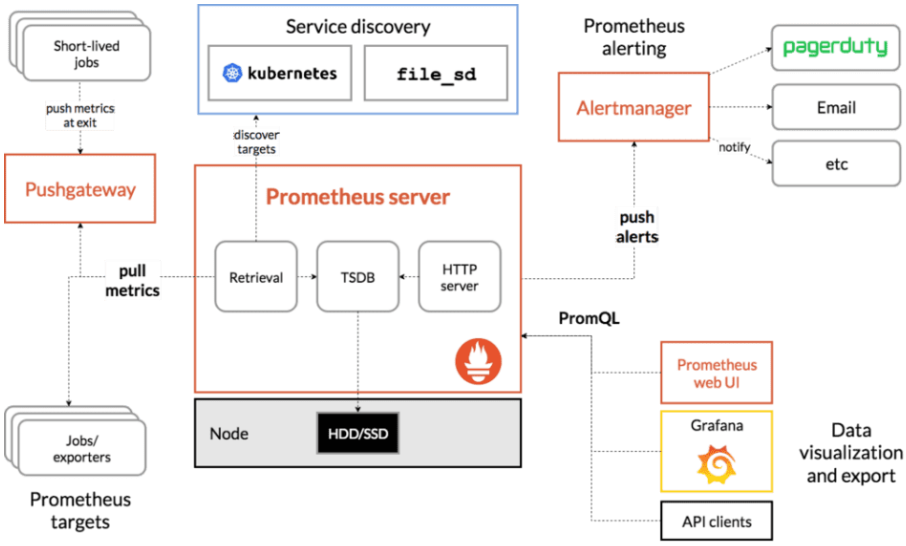
Prometheus, написаний на мові програмування go, має розподілену архітектуру. Спочатку Prometheus збирає дані за допомогою сервера Prometheus і знаходить цілі за допомогою Service discovery. Після цього зібрані дані надсилаються на інформаційну панель і обробляються за допомогою PromQL і надсилають сповіщення менеджеру сповіщень, який надсилатиме сповіщення користувачеві.

Ключові компоненти системи Prometheus:

- Сервер Prometheus – центральний компонент системи, який збирає багатовимірні дані в часових рядах, а потім аналізує та агрегує їх.
- Шлюз Prometheus – проміжне джерело, яке використовується для показників тих завдань, які не можна отримати звичайними методами. Він

діє як посередник між джерелами даних і сервером Prometheus, переводячи дані метрики у формат, зрозумілий Prometheus.

- Менеджер сповіщень – окремий компонент, який відповідає за керування сповіщеннями. Він отримує сповіщення від сервера Prometheus і керує їхнім життєвим циклом, включно з дедуплікацією і групуванням сповіщень за ступенем серйозності, а також направляє їх до правильної програми, як-от електронну пошту, PagerDuty та Slack.
- Prometheus Exporters – інструменти сторонніх розробників, які допомагають отримати показники, коли їх неможливо отримати напряму, вони зазвичай написані на Go, Python або Java та спілкуються із сервером Prometheus за допомогою протоколу HTTP.



(Рис. 1.3.1 – архітектура Prometheus)[4]

Головні особливості Prometheus:

- Багатовимірна модель даних – використання даних часових рядів, які ідентифікуються назвою метрики та парами ключ-значення.
- PromQL – гнучка мова запитів, яка може використовувати багатовимірну модель даних.
- Немає залежності від розділеного зберігання– усі окремі серверні вузли залишаються автономними.

- Pull модель – Prometheus може збирати дані часових рядів, активно «витягуючи» дані через HTTP.
- Візуалізація – Prometheus пропонує кілька типів графіків і інформаційних панелей.

1.4. Порівняльний аналіз

Zabbix, Nagios і Prometheus — популярні рішення для моніторингу інфраструктури, з відкритим кодом, які можуть контролювати стан серверів, мереж, компонентів інфраструктури, віртуальних машин тощо. Усі три системи мають високу масштабованість і забезпечують широкий спектр функцій моніторингу: збір даних, попередження, візуалізацію, налаштування та розширення. Проте, кожна система має різні сильні та слабкі сторони та задовольняє різні потреби.

Ці інструменти використовують різні стратегії для збору даних. Zabbix і Nagios є прикладами клієнт-серверної архітектури, де агенти встановлюються на контрольовані системи для збору даних і передачі їх на центральний сервер. Однак, вони також можуть підтримувати безагентний моніторинг. Zabbix дещо складніший порівняно з Nagios, оскільки вимагає зовнішнього сховища даних, наприклад MySQL або PostgreSQL. На відміну від них, Prometheus використовує PromQL для отримання та агрегування даних часових рядів із багатовимірної моделі даних у режимі реального часу, де сервер Prometheus періодично знімає показники з систем, що контролюються. Завдяки цьому Prometheus краще підходить для моніторингу хмарних систем і архітектур мікросервісів, де ресурси створюються та знищуються динамічно. Усі системи підтримують виявлення аномалій і прогнозний моніторинг, але кожна система має свій унікальний підхід.

Тип моніторингу	Zabbix	Nagios	Prometheus
Моніторинг мережі	Так	Так	Так
Серверний моніторинг	Так	Так	Так
Хмарний моніторинг	Так	Ні	Так
Системний моніторинг	Так	Так	Так
Моніторинг баз даних	Так	Ні	Так
Моніторинг IT-інфраструктури	Так	Так	Так
Моніторинг контейнерів	Так	Ні	Так
Моніторинг додатків	Так	Ні	Так

(Рис. 1.4.1 – типи моніторингу)

Ще однією подібністю є те, що всі інструменти підтримують пасивні та активні перевірки. Пасивні перевірки включають збір даних, які надсилаються на сервер моніторингу, тоді як активні перевірки передбачають надсилання запитів до контрольованої системи для збору даних. Також усі системи підтримують моніторинг SNMP, який є популярним протоколом для моніторингу мережевих пристроїв, таких як маршрутизатори та комутатори.

Prometheus виділяється як єдиний інструмент, який підтримує моніторинг JMX, що використовується для моніторингу програм Java. Крім того, тільки Prometheus підтримує хмарний out-of-the-box моніторинг, що робить його кращим вибором для організацій, які покладаються на хмарну інфраструктуру.

У свою чергу, Zabbix— єдиний інструмент, який підтримує моніторинг IPMI, який використовується для моніторингу серверного обладнання, наприклад температури та швидкості вентилятора. Zabbix також підтримує моніторинг Docker і Kubernetes, що робить його кращим вибором для організацій, які працюють з контейнерами.

Prometheus значно легше налаштувати, порівнюючи з іншими системами, а також, він також має більшу спільноту, де ви зможете отримати супровід під час налаштування.

Коли справа доходить до візуалізації даних, Zabbix має явну перевагу. На відміну від Prometheus, ви можете створювати діаграми, карти та інформаційні панелі, не покладаючись на сторонні служби, такі як Grafana. Zabbix також має краще розроблений інтерфейс користувача, краще рішення для керування користувачами та кращу функцію сповіщень.

Розглянуті системи мають свої переваги і недоліки, втім кожна система може бути оптимальним вибором для певних умов використання. Наприклад, Zabbix і Nagios орієнтовані на використання в мережах невеликих і середніх підприємств, в той час, як Prometheus – це система моніторингу для мереж великих підприємств. Зокрема, Prometheus дозволяє створювати розподілені системи моніторингу з декількома повнофункціональними вузлами, що добре узгоджується з архітектурою мережі сучасного підприємства, яка поєднує між собою географічно рознесені офіси. Втім, суттєвим недоліком Prometheus є складність цієї системи. В наступному розділі буде розглянута побудова системи моніторингу, яка підтримує структуру з множиною повнофункціональних вузлів, які можуть бути розгорнуті в географічно рознесених підмережах підприємства, але при цьому має складність, порівняну з Zabbix і Nagios.

РОЗДІЛ 2. Структурна розробка власного рішення для моніторингу

У даному розділі детально описано розробку рішення для моніторингу, яке відповідатиме наступним вимогам:

- забезпечення можливості використання в розподілених мережах;
- відмовостійкість;
- відносно невелика складність, яку можна порівняти з Zabbix та Nagios.

Після проведеного аналізу різноманітних технологій і підходів до розробки, та огляду готових компонентів моніторингу мережі, для досягнення поставленої задачі було обрано взяти за основу компоненти Zabbix і збудувати на їх основі систему з потрібними показниками.

2.1. Розподілений моніторинг у Zabbix

Розподілений мережевий моніторинг, також відомий як моніторинг децентралізованої мережі — це стратегія моніторингу, яка надає дані про продуктивність мережі на основі інформації, наданої кількома точками спостереження, які також називаються агентами моніторингу або датчиками, про конкретний контрольований об’єкт або ціль.

По суті, розподілений або децентралізований мережевий моніторинг дозволяє точно визначити проблеми мережі з окремими програмами або пристроями у мережі та визначити, чи це проблема мережі чи програми.

У розподіленій системі моніторингу кілька серверів або агентів моніторингу розгортаються в різних географічних місцях або сегментах мережі. Ці сервери збирають дані з різних пристроїв і систем і надсилають ці дані назад на центральний сервер для обробки та аналізу.

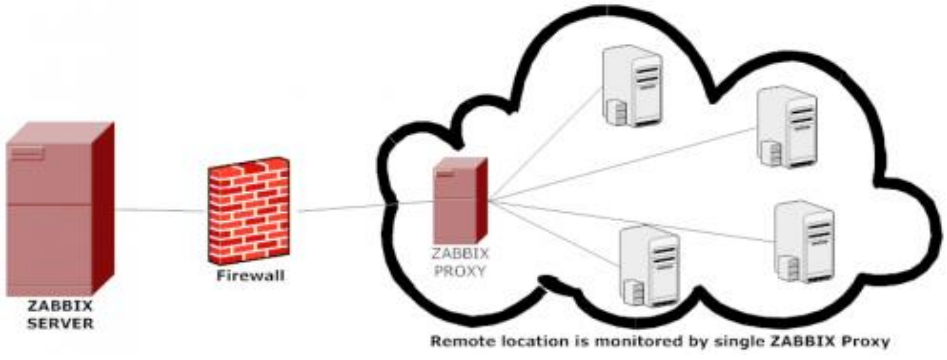
У Zabbix розподілений моніторинг може бути досягнутий за допомогою проксі-серверів Zabbix У цьому режимі проксі-сервери Zabbix діють як посередники між контрольованими хостами та центральним сервером Zabbix. Проксі-сервери збирають дані від кількох агентів Zabbix і кешують їх перед пересиланням на сервер через регулярні проміжки часу. Це зменшує мережевий трафік і навантаження на сервер, дозволяючи йому обробляти більшу кількість контрольованих хостів і елементів.

Після того як Zabbix агент з контрольованого хосту надіслав дані моніторингу до проксі-сервера Zabbix, сервер перевіряє наявність кешованих даних для цього хосту. Якщо він їх знаходить, то оновлює новими значеннями кешовані дані. Інакше, нові дані додаються до кешу. Далі через регулярні проміжки часу проксі пересилає кешовані дані на сервер Zabbix, який зберігає отримані дані від кількох проксі-серверів у базі даних. Після цього сервер обробляє дані та за допомогою них створює звіти, попередження чи сповіщення.

Крім того, використання проксі є найпростішим способом реалізації централізованого та розподіленого моніторингу, коли всі агенти та проксі звітують одному серверу Zabbix, а всі дані збираються централізовано[5].

Проксі Zabbix можна використовувати для:

- моніторингу віддалених локацій;
- моніторингу локацій з ненадійним зв’язком;
- розвантаження серверу Zabbix під час моніторингу тисяч пристроїв;
- спрощення обслуговування розподіленого моніторингу.



(Рис. 2.1.1 – архітектура розподіленого моніторингу з використанням проксі[5])

Проксі вимагає лише одного TCP-з’єднання із сервером Zabbix. Таким чином легше налаштувати обхід firewall, оскільки вам потрібно налаштувати лише одне правило для firewall.

Усі дані, зібрані проксі-сервером, зберігаються локально перед передачею на сервер. Таким чином дані не втрачаються через будь-які тимчасові проблеми зв’язку із сервером. Параметри ProxyLocalBuffer і ProxyOfflineBuffer у файлі конфігурації проксі-сервера визначають, як довго дані зберігатимуться локально.

Zabbix proxy є збирачем даних. Він не обчислює тригери, не обробляє події та не надсилає сповіщення. Щоб отримати огляд функцій проксі-сервера, було розглянуто наступну таблицю:

Функція	**Підтримка проксі**
Вбудований веб-моніторинг	Так
Попередня обробка значення елемента	Так
Автореєстрація активного агента	Так
Виявлення низькорівневих вразливостей	Так
Дистанційне керування	Так
Обчислення тригерів	Ні
Обробка подій	Ні
Кореляція події	Ні
Надсилання сповіщень	Ні

(Рис. 2.1.2 – функцій проксі-сервера)

Якщо сервер Zabbix деякий час не працював, а проксі-сервери зібрали багато даних, а потім сервер запускається, він може бути перевантажений (використання кешу історії деякий час залишається на рівні 95-100%). Це перевантаження може призвести до зниження продуктивності, коли перевірки обробляються повільніше, ніж слід.

Коли кеш історії сервера Zabbix заповнений, доступ до запису в кеш історії припиняється, призупиняючи процеси збору даних сервера. Найпоширеніший випадок перевантаження кешу історії – це після простою сервера, коли проксі-сервери завантажують зібрані дані. Щоб уникнути цього додано обмеження проксі-сервера (наразі його неможливо вимкнути).

Сервер Zabbix припинить приймати дані від проксі-серверів, коли використання кешу історії досягне 80%. Натомість ці проксі-сервери буде додано до списку обмежень. Це триватиме, доки використання кешу не знизиться до 60%. Тепер сервер почне приймати дані від проксі-серверів по черзі, визначені регулюючим списком. Це означає, що перший проксі-сервер, який намагався завантажити дані протягом періоду обмеження, буде обслуговуватися першим, і поки це не буде зроблено, сервер не прийматиме дані з інших проксі-серверів.

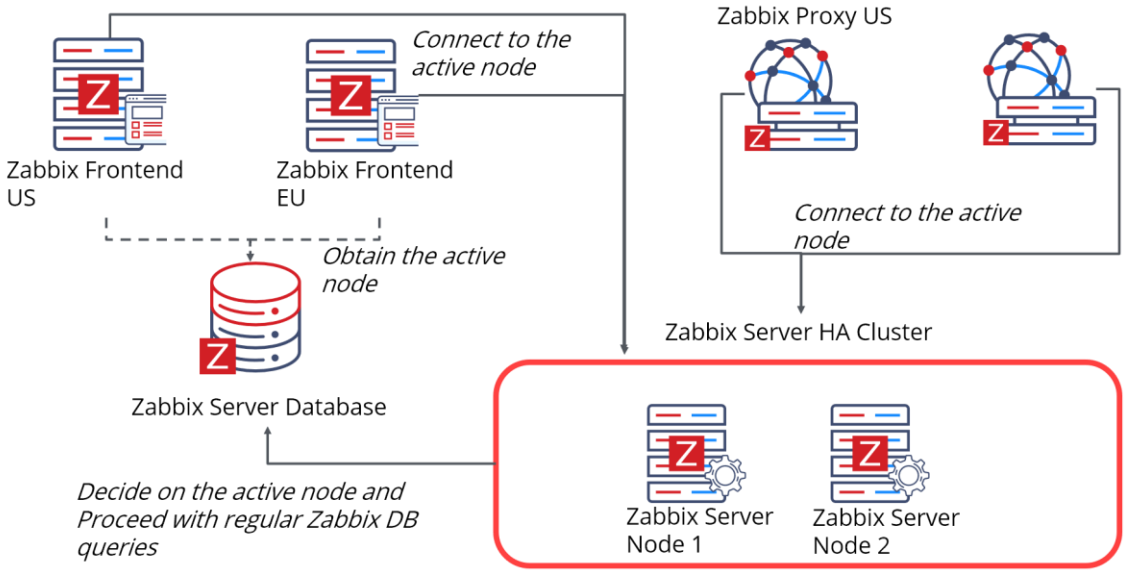
Цей режим регулювання триватиме, доки використання кешу знову не досягне 80%, або не впаде до 20%, або список регулювання не стане порожнім. У першому випадку сервер знову припинить приймати дані проксі. У двох інших випадках сервер почне нормально працювати, приймаючи дані від усіх проксі[5].

2.2. Висока доступність

Висока доступність (HA) є критично важливим аспектом будь-якої системи моніторингу, включаючи Zabbix. HA – це здатність системи працювати безперервно без збоїв протягом заданого періоду часу, тим самим зменшуючи вплив на бізнес-операції. Щоб досягти цього кілька серверів Zabbix розгортаються в кластерній архітектурі, де кожен сервер контролює набір хостів або пристроїв. Поки один сервер Zabbix у кластері активний, інші знаходяться в режимі очікування, готові взяти на себе роботу, якщо це необхідно. Усі сервери в кластері спільно використовують ту саму базу даних, у якій зберігаються дані моніторингу. Статус усіх вузлів зберігається у таблиці ha_node[6].

Кластер високої доступності сервера Zabbix підтримує необмежену кількість вузлів сервера Zabbix. Вузли повідомлятимуть про свій статус кожні 5 секунд, оновлюючи відповідний запис у таблиці ha_node. Резервні вузли відстежують час останнього доступу активного вузла з таблиці ha_node.

Якщо різниця між часом останнього доступу та поточним часом досягає затримки відновлення після відмови, кластер переходить на резервний вузол. Операція відновлення після відмови реєструється в журналі сервера Zabbix Можна визначити спеціальну затримку відновлення після відмови – часове вікно, після якого недоступний активний вузол вважається втраченим і відбувається перехід на один із резервних вузлів. Zabbix Frontend автоматично переключиться на активний вузол сервера Zabbix.



(Рис. 2.2.1 – Архітектура HA в Zabbix)[7]

В архітектурі високої доступності є 2 типи серверів:

- активні сервери: відповідають за моніторинг хостів і збір даних;
- пасивні сервери: діють як резервні сервери, які готові взяти на себе завдання моніторингу в разі збою активного сервера.

Кожен вузол:

- налаштовується окремо;
- використовує ту саму базу даних;

- може мати кілька режимів: активний, очікування, недоступний, зупинений;

Тільки один вузол може бути активним (робочим) одночасно. Резервний вузол запускає лише один процес - менеджер HA. Резервний вузол не збирає, не обробляє дані та не займається іншою звичайною діяльністю сервера (вони не слухають порти та мають мінімум підключень до бази даних).

Переваги:

- HA гарантує, що служби моніторингу залишатимуться доступними та працездатними навіть у разі збоїв апаратного чи програмного забезпечення. Це допомагає запобігти простоям і гарантує, що дані моніторингу завжди будуть доступні, коли це необхідно.
- Конфігурація високої доступності може допомогти розподілити навантаження між декількома серверами, що може призвести до підвищення продуктивності та швидшого часу відповіді.
- HA полегшує масштабування інфраструктури моніторингу в міру зростання організації. Додати нові сервери до кластера високої доступності відносно легко, і це можна зробити без переривання служб моніторингу.

Загалом, впровадження високої доступності в Zabbix може допомогти забезпечити постійну доступність і працездатність служб моніторингу, що може допомогти покращити бізнес-операції та зменшити витрати.

2.3. Підходи до управління базами даних

Від архітектури бази даних може залежати масштабованість, доступність та навіть продуктивність системи моніторингу. У Zabbix є два основні підходи до управління базами даних: база даних на сервер і спільна база даних. Нижче наведено їх переваги та недоліки, щоб обрати найкраще рішення для реалізації поставлених вимог.

База даних на сервер:

Архітектура бази даних на сервер передбачає наявність окремої бази даних для кожного сервера. Кожна база даних зберігає дані моніторингу, зібрані конкретним сервером.

Переваги:

- Масштабованість: додаткові сервери можна легко додати до системи.
- Висока відмовостійкість: так як на кожному сервері є власна база даних, якщо один з них вийде з ладу, це не завадить роботі інших серверів.
- Покращена продуктивність: даний підхід розподіляє збір і обробку даних між декількома серверами, тим самим розвантажуючи сервер Zabbix.

Недоліки:

- Складність: керування кількома базами даних може бути складним і трудомістким.
- Збільшення вартості: цей підхід вимагає більше апаратних ресурсів, оскільки кожен проксі має свою базу даних.
- Синхронізація даних: при використанні окремої бази даних для кожного серверу, може бути досить складно синхронізувати дані. Погана синхронізація, у свою чергу, призведе до невідповідності.
- Відсутність централізації: оскільки кожен сервер має власну базу даних, створити єдине уявлення про мережу може бути важко.

Спільна база даних:

У загальній базі даних кілька серверів Zabbix спільно використовують один і той же сервер бази даних. Кожен сервер Zabbix відповідає за збір даних моніторингу з призначених йому пристроїв і збереження їх у спільній базі даних.

Переваги:

- Економічно ефективний: цей підхід вимагає лише одного сервера бази даних, що зменшує витрати.

- Централізований перегляд: кілька серверів спільно використовують один сервер бази даних, що забезпечує більш централізований перегляд мережі.

Недоліки:

Низька масштабованість і доступність: оскільки спільна база даних стає єдиною точкою збою та вузьким місцем продуктивності.

- Пошкодження та неузгодженість даних: через одночасне оновлення з різних служб може збільшитися пошкодження та неузгодженість даних.
- Знижена автономія та гнучкість: оскільки усі сервери спільно використовують ті самі ресурси та дозволи бази даних.
- Складність: Управління декількома серверами Zabbix на спільному сервері бази даних може бути складним і займати багато часу.

2.4. Кластеризація бази даних

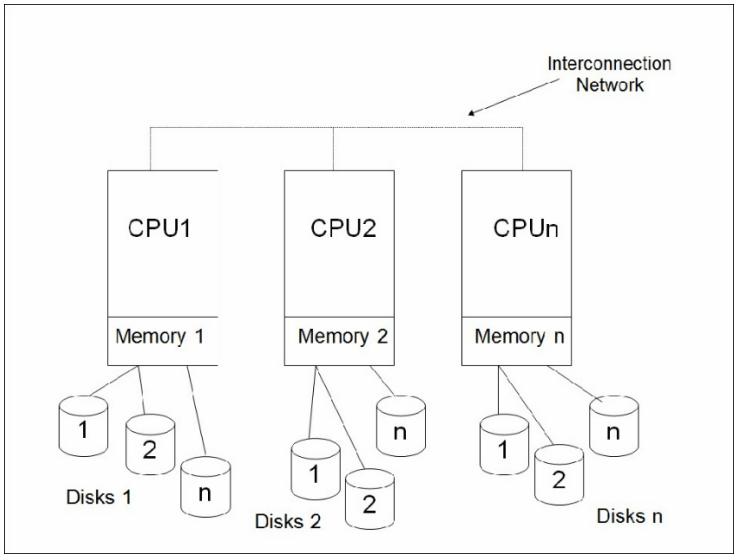
У будь-якій системі однією з основних точок збою або зниження продуктивності є база даних, тому архітектура бази є надзвичайно важливою. Прекрасним варіантом для забезпечення відмовостійкості бази даних у рішенні, яке розробляється, є кластеризація. Кластеризація бази даних — це процес об’єднання кількох серверів баз даних для спільної роботи як єдиної системи. Застосування кластеризації баз даних допомагає розвантажити систему, розподілити навантаження та проводити ремонтно-профілактичні роботи не зупиняючи систему.

Кластер даних зазвичай складається з двох або більше вузлів (серверів). Кілька серверів розподіляють робоче навантаження, забезпечуючи резервування та підвищену доступність. Це досягається шляхом тиражування даних на кількох серверах. Кожен вузол синхронізовано, тобто він матиме ті самі дані, що й усі інші вузли. Якщо один сервер виходить з ладу, дані залишаються доступними на іншому сервері.

Існує два основних типи архітектури кластеризації: архітектура shared-nothing та архітектура shared-disk.

Архітектура Shared-Nothing

У цьому режимі кластеризації бази даних кожен вузол/сервер є повністю незалежним. У цьому типі архітектури жоден окремий сервер бази даних не є головним. Це означає, що немає жодного центрального вузла бази даних, який відстежує та контролює доступ до даних у системі. Архітектура без спільного використання забезпечує чудову горизонтальну масштабованість, оскільки жодні ресурси не розподіляються між вузлами чи серверами баз даних.

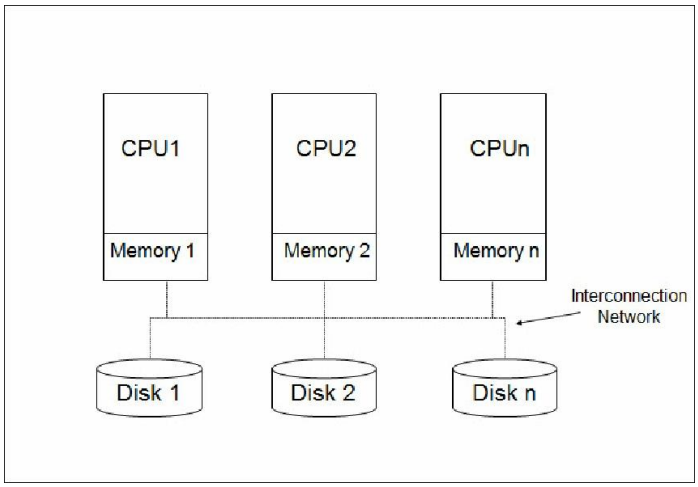


(Рис. 2.4.1 – Архітектура Shared-Nothing)[8]

Архітектура Shared-Disk

З іншого боку, ми маємо архітектуру спільного диска. У цій архітектурі всі вузли мають спільний доступ до всіх доступних серверів баз даних, згодом маючи доступ до всіх даних системи. На відміну від архітектури shared-nothing, мережевий рівень взаємозв'язку знаходиться між центральним процесором і серверами баз даних, що забезпечує доступ кількох серверів баз даних. Варто зазначити, що кластер спільного диска не пропонує великої масштабованості порівняно з архітектурою без спільного доступу, тому що, якщо всі вузли мають спільний доступ до тих самих даних, для моніторингу потоку даних у системі потрібен керуючий вузол. Проблема полягає в тому, що після перевищення певної

кількості підлеглих вузлів головний вузол не зможе ефективно контролювати та контролювати всі підлеглі вузли.



(Рис. 2.4.2. – Архітектура Shared-Disk)[8]

Різниця між цими двома архітектурами нещодавно стала розмитою з появою мережевих обчислень або розподіленого кешування. У цій конфігурації дані все ще управляються централізовано, але контролюються потужним «віртуальним сервером», що складається з багатьох серверів, які працюють разом як єдине ціле.

Переваги кластеризації баз даних:

- Масштабованість: кластерні бази даних призначені для роботи з великими обсягами даних і можуть легко масштабуватися відповідно до зростання. Оскільки потреби в моніторингу мережі розширюються, кластерна база даних може легко зростати разом з потребами, надаючи можливість продовжувати збирати й ефективно аналізувати дані.
- Висока доступність: для моніторингу розподіленої мережі потрібен високий рівень доступності, щоб забезпечити постійну доступність даних і постійну роботу моніторингу. Кластерні бази надають постійний доступ до даних, навіть у разі апаратних або програмних збоїв.
- Відмовостійкість: кластерні бази даних також пропонують відмовостійкість, тобто вони можуть продовжувати працювати навіть у разі збою мережі або

сервера. Це гарантує, що ваші дані моніторингу залишаться незмінними навіть у разі несподіваних подій.

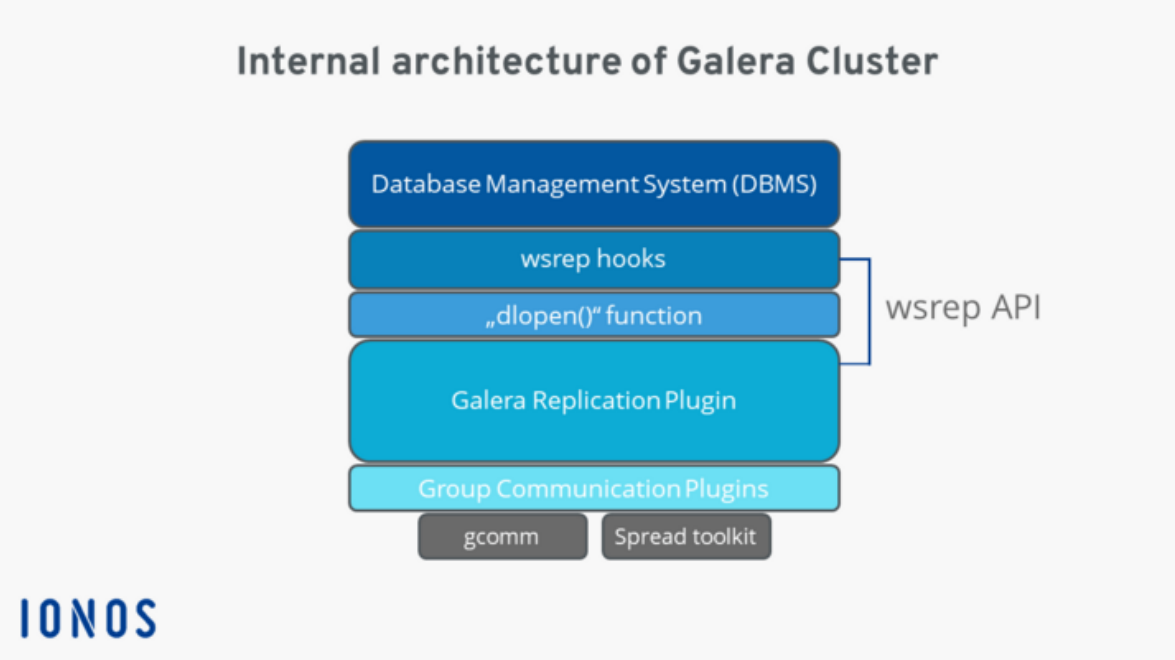
- Балансування навантаження: кластерні бази даних можна налаштувати для автоматичного розподілу навантаження між кількома вузлами, гарантуючи, що жоден вузол не буде перевантажений даними. Це допомагає підтримувати стабільну продуктивність і гарантувати, що дані моніторингу завжди збираються й аналізуються вчасно.
- Запобігання збоїв: кластер може допомогти в запобіганні довготривалих простоїв, викликаних відмовою апаратних засобів. Оскільки ресурси застосунку можуть бути швидко перекинуті на інший вузол кластера, часто навіть без втрати клієнтських підключень.
- Модернізація: нерідко виникає необхідність збільшення продуктивності серверів. Це означає необхідність міграції та тривалого простою. При використанні кластера міграцію виконати набагато легше та з мінімальним часом простою. До кластера додається новий вузол, виконується встановлення всіх необхідних оновлень. Потім за допомогою процедури переходу на резервний ресурс виконується перенесення сервера на новий вузол, а старий виключається з кластера.

2.5. Galera Cluster

Щоб забезпечити кластеризацію баз даних у даному рішенні, буде використано Galera Cluster. Galera Cluster — це синхронний кластер із кількома головними базами даних, заснований на синхронній реплікації, MySQL і InnoDB. Коли Galera Cluster використовується, читання та запис бази даних можна скеровувати на будь-який вузол. Кожен окремий вузол може бути втрачений без того, щоб це призвело до зупинки роботи системи та не потребувало використання складних процедур для відновлення після відмови.[9]

Внутрішня архітектура Galera Cluster складається з наступних чотирьох компонентів:

- Система управління базами даних: СУБД є центральною одиницею кластера. На кожному вузлі працює відповідний сервер бази даних. Galera Cluster підтримує MariaDB, MySQL і Percona XtraDB.
- API wsrep: API wsrep визначає та реалізує інтерфейс і відповідальність за доступ до підключених серверів баз даних. Він також регулює реплікацію даних. API має два основні елементи: хуки wsrep, які посилаються на сервер бази даних для реплікації даних, і dlopen(), яка є функцією, яка забезпечує зв’язок із хуками wsrep.
- Galera Replication Plugin: цей плагін реалізує API wsrep. Він забезпечує рівень сертифікації, рівень реплікації (включно з протоколом реплікації) і групову комунікаційну структуру.
- Group Communication Plugins: Galera Cluster має кілька плагінів для впровадження систем групового спілкування, таких як інструментарій Spread і gcomm. Архітектуру для цих плагінів забезпечує Group Communication Framework.



(Рис. 2.5.1 – архітектура Galera Cluster)[10]

Принцип роботи полягає в тому, що Galera Cluster використовує плагін під назвою Galera Replication Plugin, який розширює API плагіна реплікації MySQL для підтримки синхронної multi-master реплікації. Цей розширений API відомий як Write-Set Replication API або wsrep API. Через API wsrep Galera Cluster забезпечує реплікацію на основі сертифікації. Транзакція для реплікації, набір записів, містить не лише рядки бази даних для реплікації, але також містить інформацію про всі блокування, які утримувала база даних під час транзакції. Потім кожен вузол сертифікує реплікований набір записів щодо інших наборів записів у черзі програми. Потім застосовується набір записів, якщо немає конфліктуючих блокувань. У цей момент транзакція вважається завершеною, після чого кожен вузол продовжує застосовувати її до табличного простору.

Мінімальний кластер Galera складається з 3 вузлів, і рекомендується запускати з непарною кількістю вузлів. Причина полягає в тому, що якщо виникне проблема із застосуванням транзакції на одному вузлі (наприклад, проблема з мережею або машина перестане відповідати), два інші вузли матимуть кворум (тобто більшість) і зможуть продовжити транзакцію.

Однією з ключових особливостей Galera Cluster є використання реплікації на основі сертифікації. У традиційній установці реплікації MySQL головна база даних надсилає свій двійковий журнал до підлеглих баз даних, які потім застосовують журнал послідовним чином. Це може призвести до проблем із затримкою та неузгодженості даних, якщо є затримки або збої в процесі реплікації.

На відміну від цього, Galera Cluster використовує підхід на основі сертифікації, коли кожна транзакція сертифікується кожним вузлом у кластері перед тим, як її закріпити. Це гарантує, що будь-який вузол у кластері завжди має таке ж представлення бази даних як інші. Набір записів для кожної транзакції містить не лише рядки бази даних для реплікації, але також містить інформацію про всі блокування, збережені базою даних під час транзакції. Ця інформація

використовується для забезпечення відсутності конфліктуючих блокувань під час застосування набору записів до табличного простору.

Іншою важливою особливістю Galera Cluster є підтримка автоматична ініціалізація та відновлення вузлів. При додаванні нового вузла до кластера, він автоматично синхронізується з уже існуючими вузлами та стає повнофункціональним членом кластера. Якщо ж вузол виходить з ладу або переходить в автономний режим, решта вузлів у кластері продовжують нормально працювати, а несправний вузол можна замінити новим, який автоматично синхронізується з кластером.

Galera Cluster також забезпечує підтримку балансування навантаження та маршрутизації запитів. Будь-який вузол у кластері може обробляти запити на читання та запис, а балансувальник навантаження може розподіляти запити по кластеру, для забезпечення рівномірного розподілу робочого навантаження. Маршрутизація запитів використовується для того, щоб запити на запис надсилалися до правильного вузла в кластері на основі розділення даних.

Переваги:

- Відсутність затримки репліки: завдяки синхронній реплікації всі вузли в кластері синхронізуються в режимі реального часу, гарантуючи, що дані завжди оновлені та узгоджені на всіх вузлах.
- Немає втрачених транзакцій: Galera Cluster гарантує, що всі транзакції будуть зафіксовані на всіх вузлах кластера, перш ніж вважатимуться завершеними. Тому жодна транзакція не буде втрачена через збій вузла.
- Масштабованість читання: Galera Cluster забезпечує масштабованість читання, що означає, що трафік читання може розподілятися між усіма вузлами в кластері. Це досягається за допомогою балансування навантаження, яке розподіляє вхідний трафік читання між усіма доступними вузлами.
- Менші клієнтські затримки: Galera Cluster забезпечує менші затримки клієнта, що означає, що клієнтські запити обробляються швидше. Це

досягається завдяки використанню реплікації з декількома головними, що дозволяє обробляти клієнтські запити будь-яким вузлом у кластері. У результаті клієнти можуть підключатися до найближчого вузла, зменшуючи затримку, пов’язану із запитами клієнта.

- Хороша хмарна підтримка: налаштування ідеально підходить для гнучкого хмарного масштабування ресурсів бази даних. Galera Cluster також полегшує розподіл даних між різними центрами обробки даних, оскільки кожну транзакцію потрібно надіслати до кожного центру обробки даних лише один раз.

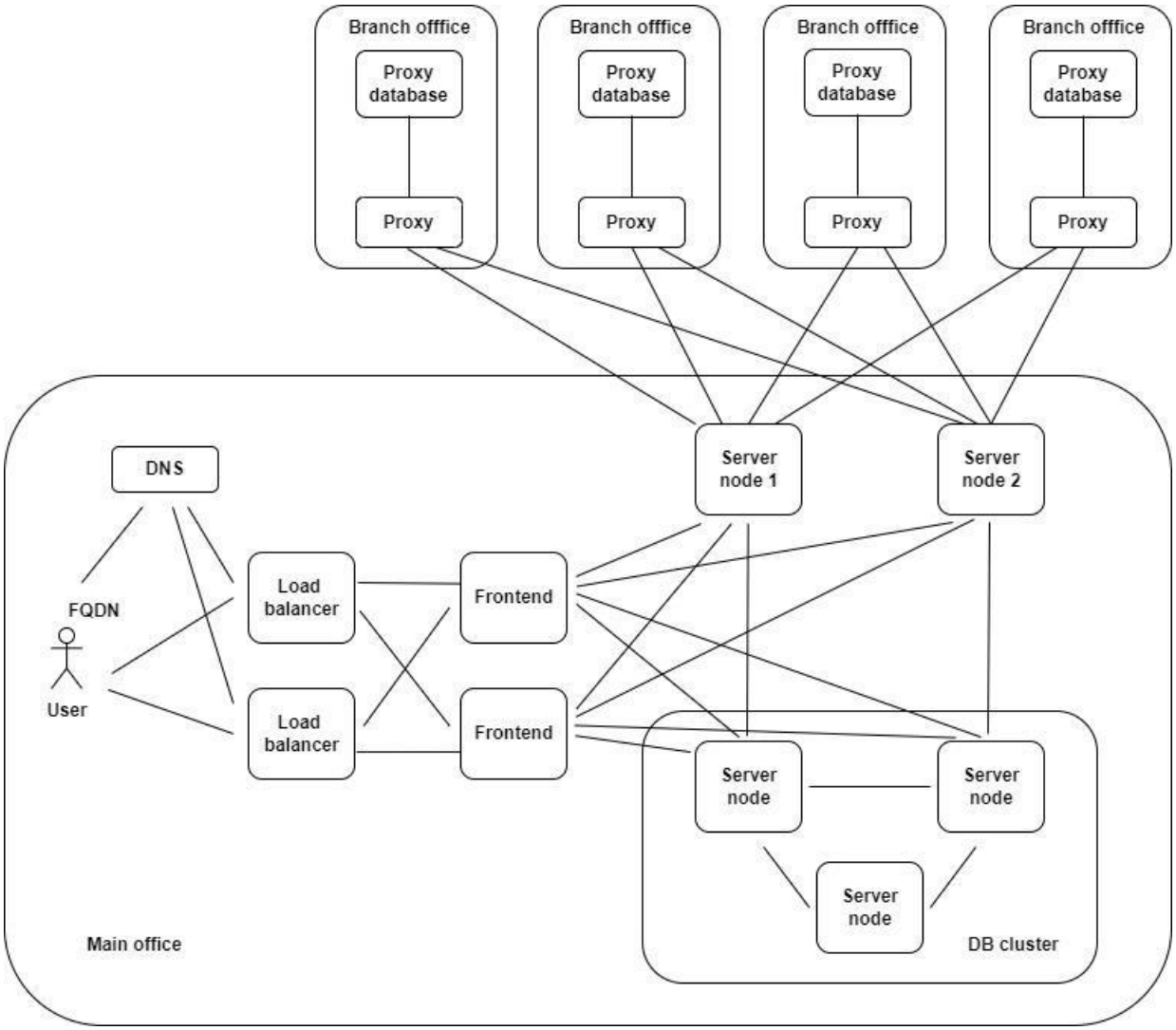
2.6. Структурна схема

Після проведеного аналізу технічних рішень було розроблено власне рішення для моніторингу.

Основні вимоги до рішення:

- Відмовостійкість: система повинна бути відмовостійкою, тобто витримувати навантаження та бути працездатною при відмові окремих компонентів, щоб забезпечити постійний доступ до інформації.
- Розподіленість: система повинна бути розподіленою, щоб забезпечити можливість працювати з інформацією з будь-якого регіонального офісу, а також зменшити час відгуку.

Для забезпечення відмовостійкості та розподіленості рішення для моніторингу підприємства було розроблено наступну архітектуру:



(Рис. 2.6.1 – структурна схема рішення для моніторингу)

Для успішного мережевого моніторингу організації з одним головним офісом і низкою регіональних необхідний комплексний підхід, який використовує низку технологій і методів моніторингу.

Щоб побудувати надійну мережеву інфраструктуру, спершу варто створити відмовостійку конфігурацію у головному офісі. Для цього головний офіс обладнаний відмовостійкою структурою з двох Zabbix-серверів, це гарантує, що якщо один сервер виходить з ладу, інший візьме на себе роботу, а мережева інфраструктура продовжить функціонувати.

Важливим компонентом мережевої архітектури будь-якого підприємства є його база даних. Збій бази даних призводить до втрати показників, і це може мати

катастрофічні наслідки для бізнесу. Щоб забезпечити постійний доступ до інформації, важливо побудувати відмовостійке сховище даних. У цьому випадку буде налаштовано MariaDB Galera Cluster. Три сервери баз даних утворюватимуть кластер і співпрацюватимуть, забезпечуючи високу доступність. Ця техніка пропонує надійну архітектуру, яка гарантує відсутність втрати даних у разі збою, оскільки дані дублюються на всіх вузлах кластера.

Користувачі вперше взаємодіють із програмою через інтерфейс, який надає їм доступ до мережевої інфраструктури. Збій фронтенду може спричинити втрату з'єднання та вплинути на взаємодію з користувачем. Тому, створення відмовостійкої конфігурації інтерфейсу має вирішальне значення. У цьому рішенні для досягнення відмовостійкості буде використано два інтерфейси. Щоб гарантувати, що користувачі завжди матимуть доступ до даних, які їм потрібні, обидва інтерфейси співпрацюватимуть. Будь-який інтерфейс зможе взяти на себе керування та продовжуватимуть обслуговувати користувачів, якщо інший вийде з ладу.

Більш того, щоб гарантувати безперервний доступ користувачів до фронтендів, перед ними будуть розміщені балансувальники навантаження. Коли користувач надсилає DNS-запит, DNS-сервер перетворює доменне ім'я на дві різні IP-адреси, по одній для кожного балансувальника навантаження. Одна з двох IP-адрес у запиті, надісланому клієнтом щодо доменного імені, отримує відповідь від сервера DNS. Коли клієнт ініціює запит до доменного імені, DNS-сервер відповідає однією з двох IP-адрес у циклічному режимі, щоб запити рівномірно розподілялися між двома балансувальниками навантаження. Зі свого боку, балансувальники навантаження рівномірно розподіляють запити користувачів між двома інтерфейсами. Це гарантує, що жоден інтерфейс не буде перевантажений. Крім того, якщо один інтерфейс виходить з ладу, балансувальники навантаження перенаправляють запити до робочого інтерфейсу, забезпечуючи відмовостійкість.

Розподілений моніторинг має вирішальне значення в мережевій інфраструктурі, що складається з кількох регіональних офісів, оскільки він дозволяє завчасно виявляти проблеми та контролювати кожен офіс. Тому, для моніторингу регіональних офісів потрібні проксі-сервери, які служать шлюзами між головним та регіональними офісами. Проксі збирає дані з регіонального офісу та надсилає їх до головного для аналізу. Це дозволяє мережевому адміністратору миттєво відстежувати стан мережі для регіонального офісу та бачити будь-які потенційні проблеми. Крім того, проксі-сервери можуть керувати трафіком і блокувати шкідливі запити, підвищуючи безпеку мережі та сприяючи ефективній роботі.

РОЗДІЛ 3. Створення програмного компоненту системи

У даному розділі покроково описано процес створення тестової конфігурації системи моніторингу. Даний компонент розроблений для демонстрації роботи різних видів моніторингу: з агентом та без.

3.1. Розгортання системи моніторингу

Для локального запуску системи моніторингу Zabbix було використано контейнеризацію Docker. Основною причиною використання контейнерів є можливість створення узгодженого та ізольованого середовища для системи. Контейнери дозволяють легко розгортати, масштабувати та керувати середовище Zabbix. Крім того, використання контейнерів Docker дозволяє запускати середовище Zabbix на будь-якій платформі, яка підтримує Docker.

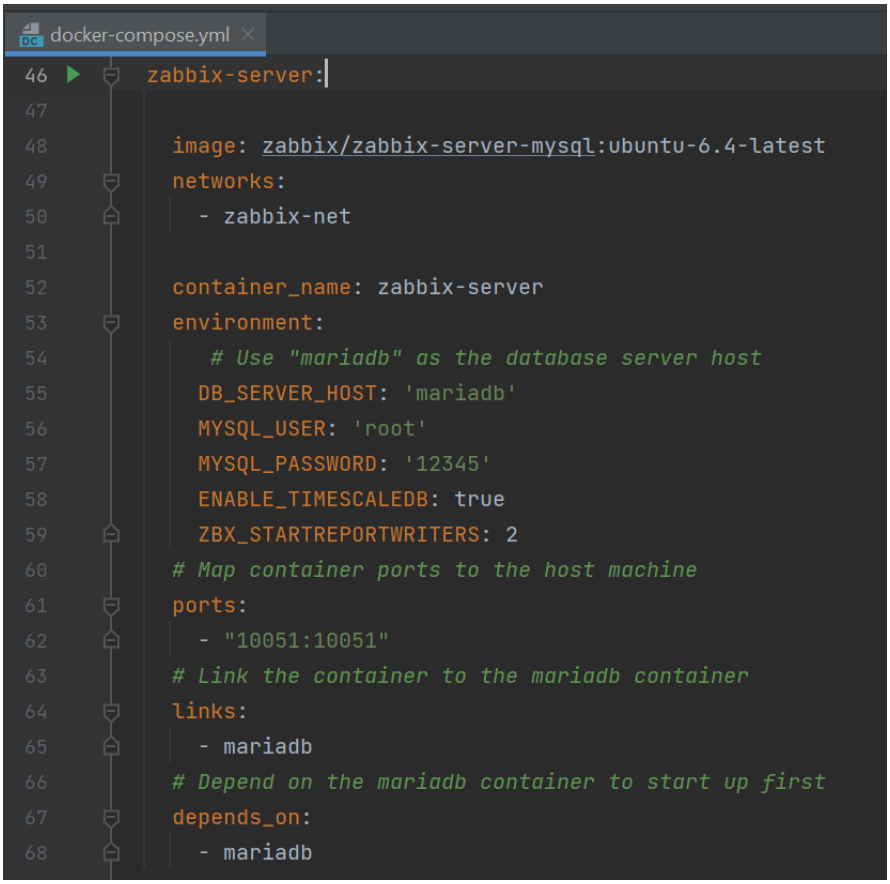
За допомогою технології Docker Compose було розгорнуто наступні сервіси: MariaDB Galera Cluster, Zabbix Server, Zabbix Agent і Zabbix Web. Сервіси взаємодіють між собою через приватну мостову мережу під назвою «zabbix-net».

Агент Zabbix відповідає за збір даних про стан різних метрик, таких як використання CPU, пам'яті, дисків, мережі тощо, та відправляє їх на сервер. Сервер Zabbix є центральною складовою системи моніторингу. Він отримує, обробляє та зберігає дані в MariaDB для подальшого використання. MariaDB використовується сервером Zabbix для зберігання всієї історії метрик, подій та конфігураційних даних. MariaDB гарантує надійне збереження цих даних та забезпечує швидкий доступ до них. Zabbix Web надає можливість переглядати зібрані дані моніторингу, керувати системою та створювати власні метрики через інтерфейс користувача.

Далі (на Рис. 3.1.1.) наведено визначення контейнеру “zabbix-server” з набором конфігураційних параметрів, які включають в себе: налаштування

підключення до бази даних, налаштування порту та мережі, яку використовує контейнер.

У налаштуваннях контейнеру “zabbix-agent” (Рис. 3.1.2.) вказані параметри для підключення до серверу Zabbix, щоб забезпечити їх взаємодію. Повний код файлу docker-compose.yml наведено в Додатку А.

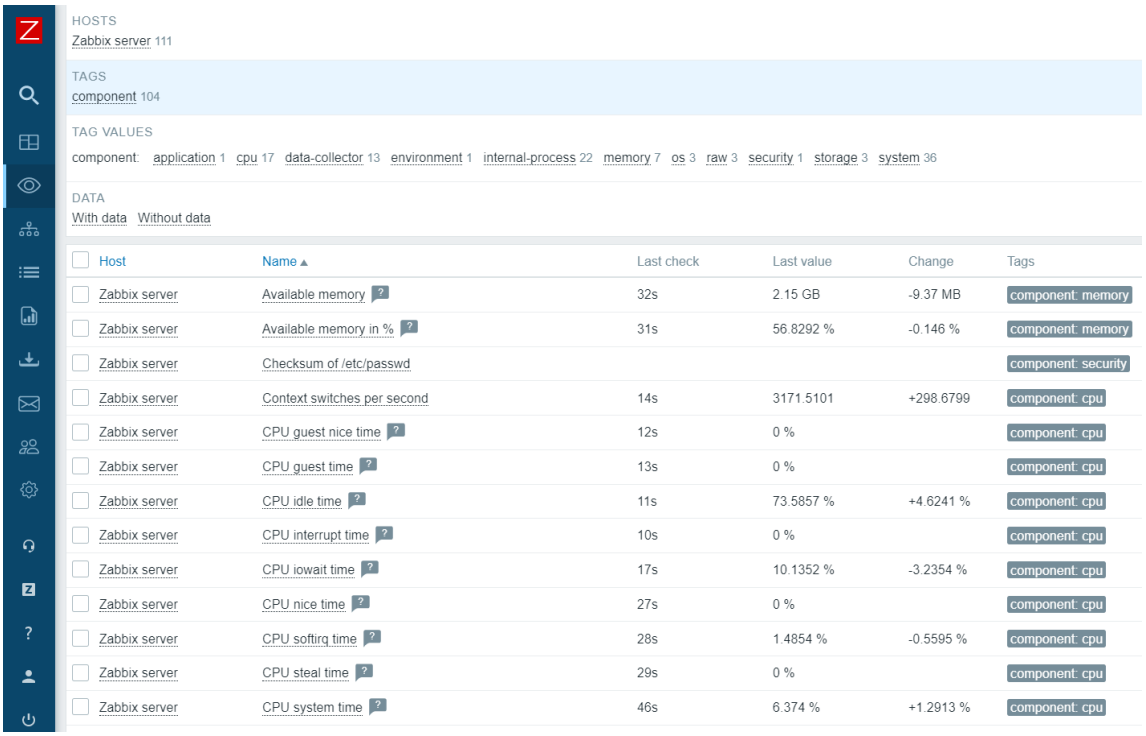


(Рис. 3.1.1. – Визначення контейнеру для Zabbix серверу в файлі docker-compose.yml)

```yaml
# Zabbix agent container for monitoring host metrics
zabbix-agent:
  image: zabbix/zabbix-agent:ubuntu-6.4-latest
  user: root
  networks:
    - zabbix-net
  # Link the container to the zabbix-server container
  links:
    - zabbix-server
  # Set privileged access mode for allowing resource access
  privileged: true
  environment:
    ZBX_HOSTNAME: Zabbix server
  ports:
    - "10050:10050"
```

(Рис. 3.1.2. – Визначення контейнеру для Zabbix агента в файлі docker-compose.yml)

Після запуску команду docker compose up для файлу docker-compose.yml, усі служби було розгорнуто та підключено одна до одної, утворюючи функціональну мережу моніторингу. Нижче наведено результат збору метрик агентом про стан системи та відображення їх у інтерфейсі Zabbix (Рис.3.1.3.).



(Рис. 3.1.3. – Панель управління зібраними метриками)

3.2. Створення застосунку автентифікації користувачів

За допомогою мови програмування Python та фреймворку FastAPI було написано RESTful застосунок автентифікації користувачів. У ньому є три кінцеві точки для реєстрації, входу та виходу користувача (Рис. 3.2.1.).

Під час реєстрації (Рис. 3.2.2.), до застосунку надсилається запит із даними користувача. Далі за допомогою SQL запиту, створюється новий запис користувача у базі даних. Якщо запит виконано успішно, функція повертає об’єкт моделі User у вигляді JSON документу.

Кінцева точка	HTTP Метод	URL	Схема

signup	POST	/user/signup	{
			"fullname": "string",
			"email": "user@example.com",
			"password": "string",
			"role": "user",
			"location": "europe"
			}

login	POST	/user/login	{
			"email": "user@example.com",
			"password": "string"
			}

logout	POST	/user/logout	{
			"email": "user@example.com",
			"password": "string"
			}

(Рис. 3.2.1. – Кінцеві точки застосунку для авторизації)

```python
# Define a POST route for user signup
@app.post("/user/signup", tags=["user"], status_code=status.HTTP_200_OK)
def user_sighup(user: User = Body(default=None)):
    try:
        # SQL query to insert a new user into the database
        sql = """INSERT INTO users (fullname, email, password, role, location)
                    VALUES (%s, %s, %s, %s, %s)"""
        val = (user.fullname, user.email, user.password, user.role.value, user.location.value)
        run_query(sql, val, is_update=False)
        return user.json()
    except Exception as error:
        # If an error occurs during the signup process, return a JSON response with an error message
        return Response(content={"error": str(error)}, media_type="application/json",
                        status_code=status.HTTP_500_INTERNAL_SERVER_ERROR)
```

(Рис. 3.2.2. – Функція для реєстрації користувачів)

Під час входу користувача в систему (Рис. 3.2.3.), за допомогою SQL запиту перевіряється наявність користувача в базі даних. Якщо даного користувача знайдено, його статус змінюється з неактивного на активний (Рис. 3.2.4.). Вихід користувача з системи виконується за аналогією до входу, лише статус повертається до неактивного стану.

```python
# Define a POST route for user login
@app.post("/user/login", tags=["user"], status_code=status.HTTP_200_OK)
def user_login(user: UserLogin = Body(default=None)):
    # check if the user exists in the system
    if check_user(user):
        # set the user's status as active
        set_active(user, True)
        return user.json()
    else:
        # If the user credentials are invalid, raise an HTTPException with a 403 Forbidden status code and error message
        raise HTTPException(status_code=status.HTTP_403_FORBIDDEN, detail={"error": "Invalid login details!"})
```

(Рис. 3.2.3. – Функція для входу користувачів)

```python
# function to set the active status of a user in the database
def set_active(user: UserLoginSchema, value: bool):
    query = """UPDATE users
        SET is_active = %s
        where email=%s and password=%s"""
    val = (int(value), user.email, user.password)
    run_query(query, val, is_update=True)
```

(Рис. 3.2.4. – Функція для зміни статусу користувача)

Для зберігання інформації про користувачів було розгорнуто контейнер “mysql” (Рис. 3.2.6.), який містить базу даних MySQL. У базі створено таблицю users з наступними полями:

Поле	Тип даних
fullname	varchar(20)
email	varchar(40)
password	varchar(20)
role	varchar(10)
location	varchar(15)
is_active	tinyint(1), default = 0

(Рис. 3.2.5. – Схема таблиці users)

Для збереження інформації про локацію та роль користувача використовуються класи типу enum, які містять у собі набори доступних значень (Додаток В).

```
mysql:
  image: mysql:5.7
  environment:
    MYSQL_DATABASE: 'user'
    MYSQL_USER: 'mysql'
    MYSQL_PASSWORD: 'root'
    MYSQL_ROOT_PASSWORD: 'root'
  ports:
    - '3307:3306'
  expose:
    - '3306'
  volumes:
    - mysql_data_container:/var/lib/mysql
```

(Рис. 3.2.6. – Визначення mysql контейнеру)

3.3. Імітація трафіку

Щоб імітувати трафік у системі, було створено Python сценарій, який виконує дії входу та виходу для випадкової кількості користувачів протягом певного проміжку часу.

Для цього було визначено функцію mock_traffic (Додаток Ґ), яка є вхідною точкою для програми створення трафіку. Вона неодноразово викликає функції login_users і logout_users, тим самим створюючи навантаження на застосунок. Функція login_users імітує дії входу для певної кількості користувачів. Дана функція отримує список неактивних користувачів із бази даних за допомогою додаткової функції get_active_users. Для кожного користувача створюється об’єкт UserLogin з електронною адресою і паролем користувача та надсилає запит POST до кінцевої точки /user/login системи. Він перевіряє код статусу відповіді, щоб визначити, чи вхід був успішним чи ні. Функція logout_users працює за аналогією, намагаючись зробити певну кількість користувачів неактивними за допомогою кінцевої точки /user/logout.

У коді використовується генерація випадкових чисел із вказаного діапазону, яка додається до вказаної кількості користувачів під час дій входу та виходу. Це допомагає імітувати різні рівні трафіку в системі.

3.4. Реалізація моніторингу

Щоб моніторити створений трафік було розроблено Python сценарій, який збирає та надсилає метрики до серверу Zabbix (Додаток Д).

Через веб-інтерфейс Zabbix було створено метрики для збору та відстеження показників пов’язаних з активність користувачів (Рис. 3.4.1.), таких як активні клієнти системи, активні користувачі з певною роллю та кількість клієнтів із певних регіонів.

Даний скрипт взаємодіє з базою даних, дістаючи необхідні показники для метрик через запит SQL. Також, сценарій містить клас, який представляє

налаштування для моніторингу за допомогою Zabbix. Він має конструктор, який встановлює необхідні значення імені хоста для сервера Zabbix і локального агента. Метод send_metric у класі ZabbixMonitoring надсилає метрику на сервер Zabbix. Він використовує клас ZabbixSender з бібліотеки pyzabbix для встановлення з'єднання з агентом Zabbix. Метод створює об'єкт ZabbixMetric, що містить ім'я хоста Zabbix, ім'я метрики (представлене значенням із переліку Metrics (Додаток В) і значення самої метрики. Потім надсилаючи дані на сервер Zabbix із заданою періодичністю для кожної метрики.

```
# define Metrics enum with different metrics that can be used to evaluate the system
class Metrics(enum.Enum):
    TOTAL_NUMBER_OF_ACTIVE_CLIENTS = "total_number_of_active_clients"
    TOTAL_NUMBER_OF_ACTIVE_USERS = "total_number_of_active_users"
    TOTAL_NUMBER_OF_ACTIVE_ADMINS = "total_number_of_active_admins"
    NUMBER_OF_CLIENTS_FROM_EUROPE = "number_of_clients_from_europe"
    NUMBER_OF_CLIENTS_FROM_ASIA = "number_of_clients_from_asia"
    NUMBER_OF_CLIENTS_FROM_NORTH_AMERICA = "number_of_clients_from_north_america"
```

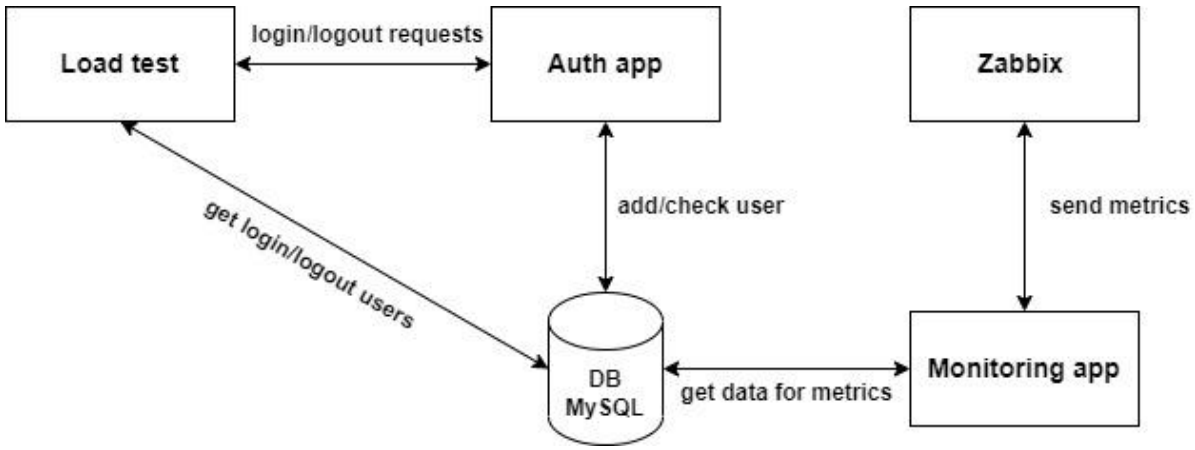
(Рис. 3.4.1. – Перелік метрик)

3.5. Результати

Було створено компонент системи, який демонструє різні можливості моніторингу: через Zabbix агент та за допомогою модуля pyzabbix.

Для цього локально було розгорнуто систему моніторингу Zabbix, яка за допомогою Zabbix агента збирає інформацію про поточний стан хоста та відправляє її до серверу.

Розроблено застосунок (Рис. 3.5.1.), який збирає метрики про активність користувачів на основі певних атрибутів і, також, відправляє їх до серверу. Даний застосунок складається 3 компонентів: автентифікації користувачів, імітації трафіку та моніторингу. Компонент моніторингу запускається одночасно з автентифікацією користувачів (Рис. 3.5.2.).



(Рис. 3.5.1. – Схема застосунку)

```python
if __name__ == '__main__':
    # For each monitoring method in the monitoring_methods list, create a new Thread object and start it
    for monitoring_method in monitoring_methods:
        thread = Thread(target=monitoring_method, daemon=True)
        thread.start()
    # Start the app using uvicorn on localhost and port 8001
    uvicorn.run(app, host="localhost", port=8001)
```

(Рис. 3.5.2. – Запуск застосунку авторизації та паралельний запуск моніторингу)

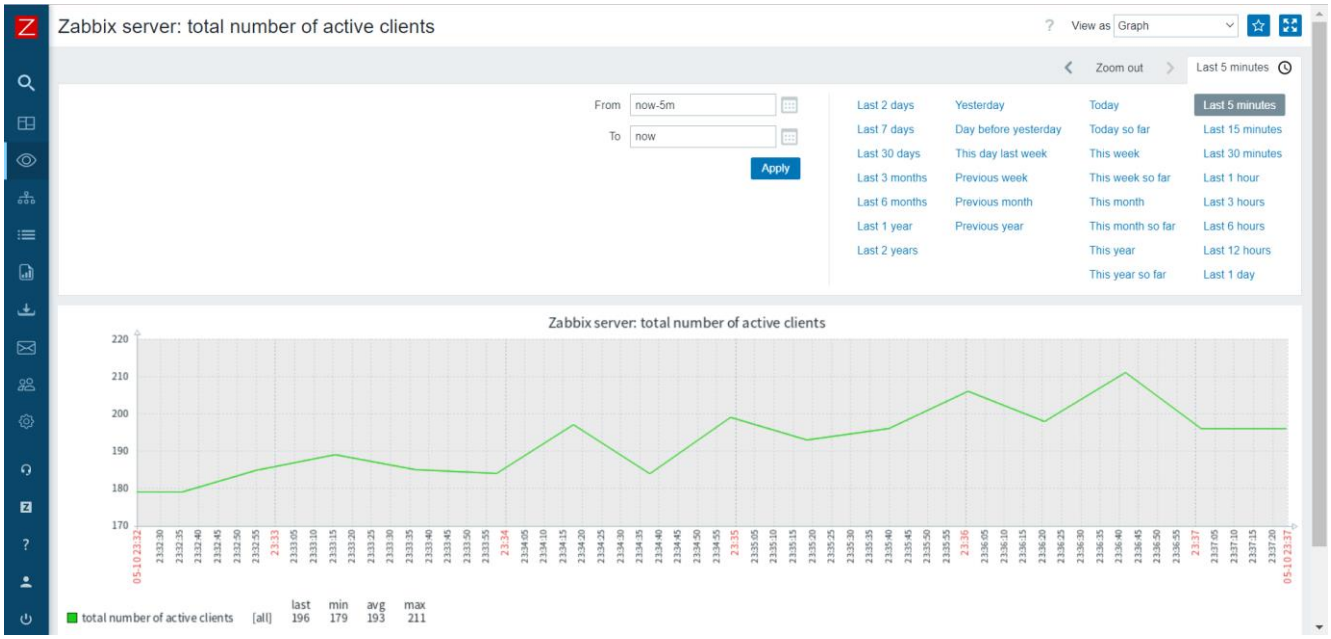
Для кожної метрики було створено метод в класі моніторингу (Рис.3.5.3.). Кожен із визначених методів, запускається в окремому потоці, що дозволяє виконувати моніторинг декількох частин застосунку одночасно.

```python
# Create a list of monitoring methods to be executed
monitoring_methods = [
    ZabbixMonitoring().collect_total_clients,
    ZabbixMonitoring().collect_total_users,
    ZabbixMonitoring().collect_total_admins,
    ZabbixMonitoring().collect_total_clients_from_europe,
    ZabbixMonitoring().collect_total_clients_from_asia,
    ZabbixMonitoring().collect_total_clients_from_north_america
]
```

(Рис. 3.5.3. – Запуск застосунку авторизації та паралельний запуск моніторингу)

Під час моніторингу, значення метрик надсилаються до Zabbix сервера, та зберігаються в базі даних MariaDB. Результат моніторингу було перевірено та візуалізовано в вигляді графіків, за допомогою інтерфейсу користувача Zabbix

Web (Рис. 3.5.4.).



(Рис. 3.5.4. – Результат моніторингу)

Висновки

У ході виконання даної кваліфікаційної роботи було досягнуто поставленої мети та виконано наступне:

- Після отримання теми та аналізу необхідності моніторингу для мережі підприємства, було сформовано задачу, яка полягає в структурній розробці системи для моніторингу мережі підприємства, яка складається з головного кампусу та територіально рознесених офісів. Система моніторингу повинна мати високу доступність за рахунок стійкості до відмов окремих компонентів.
- Сформований список існуючих прототипів до рішення, до нього увійшли Zabbix, Nagios та Prometheus, та виконаний їх детальний аналіз та порівняння в ході якого було обрано взяти за основу компоненти Zabbix як бази для побудови власного рішення.
- Проведений аналіз технічних рішень, які дозволяють реалізувати поставлені вимоги. Та на основі цього була створена структурна схема власного рішення для моніторингу. Мережа сучасного підприємства є розподіленою і складається з головного та регіональних офісів. У головному офісі створюється відмовостійка конфігурація з двох серверів, які функціонують по схемі active-standby, та базою даних у вигляді кластеру. Для відмовостійкості система також містить 2 фронтенда. Доступ до яких здійснюється через балансувальники навантаження. Моніторинг в регіональних офісах здійснюється за допомогою проксі.
- Розроблено тестову конфігурацію системи щоб продемонструвати роботу моніторингу. Для цього за допомогою технології контейнеризації Docker було розгорнуто наступні сервіси: MariaDB Galera Cluster, Zabbix Server, Zabbix Web та агент Zabbix.

Агент Zabbix збирає дані моніторингу та відправляє їх на сервер. Сервер Zabbix отримує, обробляє та зберігає ці дані в MariaDB, а Zabbix Web дозволяє переглядати зібрані дані та керувати ними.

- За допомогою мови програмування Python створено застосунок для автентифікаціїї користувачів з використанням фреймворку FastApi. У ньому є три кінцеві точки для реєстрації, входу та виходу користувача. Дані про користувачів зберігаються у базі даних MySQL. До цього застосунку, був написаний сценарій, який імітує трафік для функцій login/logout, надсилаючи HTTP-запити на локальний сервер. Також було створено Python сценарій для паралельного моніторингу та збору показників про даний трафік, який надсилає зібрані метрики до Zabbix.

Список використаних джерел

1. Zabbix- A Simpler way of Monitoring [Електронний ресурс] – Режим доступу до ресурсу:

https://medium.com/@shashankrock24/zabbix-a-simpler-way-of-monitoring-f233f3206c42

2. Nagios Documentation [Електронний ресурс] – Режим доступу до ресурсу:

https://www.nagios.org/documentation/

3. Nagios Tutorial [Електронний ресурс] – Режим доступу до ресурсу:

https://www.javatpoint.com/nagios

4. What is Prometheus? [Електронний ресурс] – Режим доступу до ресурсу:

https://prometheus.io/docs/introduction/overview/

5. Proxies [Електронний ресурс] – Режим доступу до ресурсу:

https://www.zabbix.com/documentation/current/en/manual/distributed_monitoring/proxies

6. High availability [Електронний ресурс] – Режим доступу до ресурсу:

https://www.zabbix.com/documentation/current/en/manual/concepts/server/ha

7. WHAT'S NEW IN ZABBIX 6.0 LTS BY ARTŪRS LONTONS / ZABBIX SUMMIT ONLINE 2021 [Електронний ресурс] – Режим доступу до ресурсу:

https://noise.getoto.net/2021/12/10/whats-new-in-zabbix-6-0-lts-by-arturs-lontons-zabbix-summit-online-2021/

8. What is Database Clustering? [Електронний ресурс] – Режим доступу до ресурсу:

https://www.harperdb.io/post/what-is-database-clustering

9. Overview of Galera Cluster [Електронний ресурс] – Режим доступу до ресурсу:

https://galeracluster.com/library/documentation/overview.html

10. Galera Cluster for MariaDB [Електронний ресурс] – Режим доступу до ресурсу:

https://www.ionos.com/digitalguide/hosting/technical-matters/mariadb-galera-clusters/

Додатки

Додаток А

Docker compose файл для розгортання системи моніторингу.

```
version: '2.1'

#This Docker file is used to set up a Zabbix monitoring environment
#consisting of multiple services: MariaDB, Zabbix Server, Zabbix Agent,
#and Zabbix Web. The services are connected via a bridge network called 'zabbix-net'

# Define a bridge network for the containers to communicate with each other
networks:
 zabbix-net:
   driver: bridge

services:
# MariaDB container for database storage
 mariadb:
   # Connect the container to the zabbix-net network
   networks:
     - zabbix-net
   # Use the mariadb-galera image from Docker Hub
   image: docker.io/bitnami/mariadb-galera:10.11
   # Map container ports to the host machine
   ports:
     - '3306:3306'
     - '4444:4444'
     - '4567:4567'
     - '4568:4568'
   # Persist data in the mariadb_galera_data volume
   volumes:
     - mariadb_galera_data:/bitnami/mariadb


   environment:
     - MARIADB_ROOT_PASSWORD=12345
     - MARIADB_GALERA_MARIABACKUP_PASSWORD=12345
     - MARIADB_GALERA_CLUSTER_ADDRESS=gcomm://
     - MARIADB_CHARACTER_SET=utf8mb4
     - MARIADB_COLLATE=utf8mb4_bin

   healthcheck:
     test: ['CMD', '/opt/bitnami/scripts/mariadb-galera/healthcheck.sh']
     interval: 15s # how often to perform the health check
     timeout: 5s   # how long to wait for the health check to complete
     retries: 6    # how many times to retry the health check if it fails


 # Zabbix server container for monitoring
 zabbix-server:
   image: zabbix/zabbix-server-mysql:ubuntu-6.4-latest
   networks:
     - zabbix-net
   container_name: zabbix-server
   environment:
      # Use "mariadb" as the database server host
     DB_SERVER_HOST: 'mariadb'
     MYSQL_USER: 'root'
     MYSQL_PASSWORD: '12345'
```

```
    ENABLE_TIMESCALEDB: true
    ZBX_STARTREPORTWRITERS: 2
  # Map container ports to the host machine
  ports:
    - "10051:10051"
  # Link the container to the mariadb container
  links:
    - mariadb
  # Depend on the mariadb container to start up first
  depends_on:
    - mariadb

# Zabbix agent container for monitoring host metrics
zabbix-agent:
  image: zabbix/zabbix-agent:ubuntu-6.4-latest
  user: root
  networks:
    - zabbix-net
  # Link the container to the zabbix-server container
  links:
    - zabbix-server
  # Set privileged access mode for allowing resource access
  privileged: true
  environment:
    ZBX_HOSTNAME: Zabbix server
  ports:
    - "10050:10050"

# Zabbix web interface container for viewing monitoring data
zabbix-web:
  image: zabbix/zabbix-web-apache-mysql:ubuntu-6.4-latest
  ports:
    - "80:8080"
    - "443:8443"

  networks:
    - zabbix-net
  container_name: zabbix-web
  environment:
    DB_SERVER_HOST: 'mariadb'
    MYSQL_USER: 'root'
    MYSQL_PASSWORD: '12345'
    ZBX_SERVER_HOST: 'zabbix-server'
    PHP_TZ: Europe/Kiev
  # Set up links between the container and the mariadb and zabbix-server containers
  links:
    - mariadb
    - zabbix-server
  depends_on:
    - mariadb
    - zabbix-server

mysql:
  image: mysql:5.7
  environment:
    MYSQL_DATABASE: 'user'
    MYSQL_USER: 'mysql'
    MYSQL_PASSWORD: 'root'
    MYSQL_ROOT_PASSWORD: 'root'
  ports:
```

```
      - '3307:3306'
    expose:
      - '3306'
    volumes:
      - mysql_data_container:/var/lib/mysql

volumes:
 mariadb_galera_data:
 mysql_data_container:
```

Додаток Б

Python код, що визначає RESTful API з трьома кінцевими точками для обробки запитів на реєстрацію, вхід і вихід користувачів із системи.

```
"""
This code defines a RESTful API with three routes
to handle user signup, login, and logout requests.
"""
from fastapi import FastAPI, Body, HTTPException, status
from fastapi.openapi.models import Response
from app.schema import User, UserLogin
from app.utils import run_query, check_user, set_active

app = FastAPI()

# Define a POST route for user signup
@app.post("/user/signup", tags=["user"], status_code=status.HTTP_200_OK)
def user_sighup(user: User = Body(default=None)):
    try:
        # SQL query to insert a new user into the database
        sql = """INSERT INTO users (fullname, email, password, role, location)
                    VALUES (%s, %s, %s, %s, %s)"""
        val = (user.fullname, user.email, user.password, user.role.value,
user.location.value)
        run_query(sql, val, is_update=False)
        return user.json()
    except Exception as error:
        # If an error occurs during the signup process, return a JSON response with an error
message and status code 500
        return Response(content={"error": str(error)}, media_type="application/json",
                        status_code=status.HTTP_500_INTERNAL_SERVER_ERROR)

# Define a POST route for user login
@app.post("/user/login", tags=["user"], status_code=status.HTTP_200_OK)
def user_login(user: UserLogin = Body(default=None)):
    # check if the user exists in the system
    if check_user(user):
        # set the user's status as active
        set_active(user, True)
        return user.json()
    else:
        # If the user credentials are invalid, raise an HTTPException with a 403 Forbidden
status code and error message
        raise HTTPException(status_code=status.HTTP_403_FORBIDDEN, detail={"error": "Invalid
login details!"})

# Define a POST route for user logout
@app.post("/user/logout", tags=["user"], status_code=status.HTTP_200_OK)
def user_logout(user: UserLogin = Body(default=None)):
```

```
    if check_user(user):
        # set the user's status as inactive
        set_active(user, False)
        return user.json()
    else:
        raise HTTPException(status_code=status.HTTP_403_FORBIDDEN, detail={"error": "Logout
failed"})
```

Додаток В

Цей код визначає три окремі класи enum (Role,Location та Metrics) які використовуються для визначення та забезпечення виконання наборів констант із фіксованим набором можливих значень.

```
"""
This code defines three separate enum classes (Role, Location, and Metrics)
which are used to define and enforce sets of constants with a fixed set of possible values.
"""
import enum

# define Role enum with two possible values
class Role(enum.Enum):
    USER = "user"
    ADMIN = "admin"
# define Location enum with three possible values
class Location(enum.Enum):
    EUROPE = "europe"
    ASIA = "asia"
    NORTH_AMERICA = "north america"
# define Metrics enum with different metrics that can be used to evaluate the system
class Metrics(enum.Enum):
    TOTAL_NUMBER_OF_ACTIVE_CLIENTS = "total_number_of_active_clients"
    TOTAL_NUMBER_OF_ACTIVE_USERS = "total_number_of_active_users"
    TOTAL_NUMBER_OF_ACTIVE_ADMINS = "total_number_of_active_admins"
    NUMBER_OF_CLIENTS_FROM_EUROPE = "number_of_clients_from_europe"
    NUMBER_OF_CLIENTS_FROM_ASIA = "number_of_clients_from_asia"
    NUMBER_OF_CLIENTS_FROM_NORTH_AMERICA = "number_of_clients_from_north_america"
```

Додаток Г

Цей файл визначає дві моделі Pydantic, User і UserLogin. Дані моделі використовуються для визначення схеми даних для входу, виходу та реєстрації користувачів.

```
"""
This file defines two Pydantic models, User and UserLogin.
The User and UserLogin models are used to define the data schema
for user information and user login information, respectively.
"""
from pydantic import BaseModel, Field, EmailStr
from app.enums import Location, Role

# define User model to represent the user data
class User(BaseModel):
    fullname: str = Field(default=None)
    email: EmailStr = Field(default=None)
    password: str = Field(default=None)
```

```
    # role and location fields use the Role and Location enum defined in enums.py
    role: Role = Field(default=None)
    location: Location = Field(default=None)



# define UserLogin model to represent the user login data
class UserLogin(BaseModel):
    email: EmailStr = Field(default=None)
    password: str = Field(default=None)
```

Додаток Ґ

Даний код імітує трафік у системі авторизації користувачів шляхом виконання дій входу та виходу.

```
"""
The code provided simulates traffic in a system by performing login and logout actions
for a specified duration.
"""

import requests
import datetime
from app.schema import UserLogin
import time
import random
from app.utils import run_query

# utility function to get active users
def get_active_users(is_active):
    query = "SELECT fullname, email, password, role, location FROM users WHERE is_active=%s"
    return run_query(query, (is_active,), False)
# function to simulate traffic
def mock_traffic():
    start_time = datetime.datetime.now()
    end_time = start_time + datetime.timedelta(minutes=5)
    count = 1
    while datetime.datetime.now() < end_time:
        print("...")
        login_users(count)
        logout_users(count)
        count += random.randint(0, 4)
        time.sleep(5)
# function to simulate login actions for a specified number of users
def login_users(count):
    user_list = get_active_users(is_active=0)
    if user_list:
        for i in range(count + random.randint(0, 10)):
            _, email, password, _, _ = user_list[i]
            user = UserLogin(email=email, password=password)
            response = requests.post("http://localhost:8001/user/login", user.json())
            if response.status_code == 200:
                print("login")
            else:
                print("login error")


# function to simulate logout actions for a specified number of users
def logout_users(count):
    user_list = get_active_users(is_active=1)
```

```
    if user_list:
        for i in range(count + random.randint(0, 7)):
            _, email, password, _, _ = user_list[i]
            user = UserLogin(email=email, password=password)
            response = requests.post("http://localhost:8001/user/logout", user.json())
            if response.status_code == 200:
                print("logout")
            else:
                print("logout error")

if __name__ == '__main__':
    mock_traffic()
```

Додаток Д

Сценарій Python, який налаштовує моніторинг для системи з користувачами, розташованими в різних регіонах. Сценарій використовує інструмент моніторингу Zabbix для збору метрик про активних клієнтів, активних користувачів,
активних адміністраторів і кількість клієнтів з окремих регіонів.

```
"""
This code is a Python script that sets up monitoring for a system with users located in
different regions.
The script uses the Zabbix monitoring tool to collect metrics about active clients, active
users,
active admins, and the number of clients from specific regions.
"""
import datetime
import time
from pyzabbix import ZabbixSender, ZabbixMetric
from app.utils import run_query
from app.enums import Role, Location, Metrics

# the total number of clients who are active
def get_login_clients():
    query = "SELECT COUNT(*) FROM users WHERE is_active = 1"
    return run_query(query, [], False)
# the number of clients who are active and have a specific role
def get_login_clients_by_role(role):
    query = "SELECT COUNT(*) FROM users WHERE role=%s AND is_active = 1"
    return run_query(query, (role,), False)
# the number of clients who are active and located in a specific region
def get_login_clients_by_location(location):
    query = "SELECT COUNT(*) FROM users WHERE location=%s AND is_active = 1"
    return run_query(query, (location,), False)

# Define a class for Zabbix monitoring
class ZabbixMonitoring:
    # Constructor function to initialize the class with the necessary hostname values
    def __init__(self):
        self.zabbix_hostname = 'Zabbix server'
        self.agent_hostname = 'localhost'

    # Function to send a metric to Zabbix server
    def send_metric(self, metric_name: Metrics, metric_value):
        zbx = ZabbixSender(self.agent_hostname)
        metric = [ZabbixMetric(self.zabbix_hostname, metric_name.value, metric_value), ]
```

```
        print(f"{metric_name.value} {zbx.send(metric)}")

    # Function to collect the total number of active clients and send it as a metric to
Zabbix server
    def collect_total_clients(self):
        # Set the start and end time for monitoring
        start_time = datetime.datetime.now()
        end_time = start_time + datetime.timedelta(minutes=6)
        while datetime.datetime.now() < end_time:
            print("Monitoring clients...")
            num_list = get_login_clients()
            num_active_clients = num_list[0][0]
            self.send_metric(Metrics.TOTAL_NUMBER_OF_ACTIVE_CLIENTS, num_active_clients)
            # Wait for 20 seconds before monitoring again
            time.sleep(20)

    # Function to collect the total number of active users and send it as a metric to Zabbix
server
    def collect_total_users(self):
        start_time = datetime.datetime.now()
        end_time = start_time + datetime.timedelta(minutes=6)
        while datetime.datetime.now() < end_time:
            print("Monitoring users...")
            num_list = get_login_clients_by_role(Role.USER.value)
            num_active_users = num_list[0][0]
            self.send_metric(Metrics.TOTAL_NUMBER_OF_ACTIVE_USERS, num_active_users)
            time.sleep(15)

    # Function to collect the total number of active admins and send it as a metric to
Zabbix server
    def collect_total_admins(self):
        start_time = datetime.datetime.now()
        end_time = start_time + datetime.timedelta(minutes=6)
        while datetime.datetime.now() < end_time:
            print("Monitoring admins...")
            num_list = get_login_clients_by_role(Role.ADMIN.value)
            num_active_admins = num_list[0][0]
            self.send_metric(Metrics.TOTAL_NUMBER_OF_ACTIVE_ADMINS, num_active_admins)
            time.sleep(15)

    # Function to collect the total number of active clients from Europe and send it as a
metric to Zabbix server
    def collect_total_clients_from_europe(self):
        start_time = datetime.datetime.now()
        end_time = start_time + datetime.timedelta(minutes=6)
        while datetime.datetime.now() < end_time:
            print("Monitoring europe...")
            num_list = get_login_clients_by_location(Location.EUROPE.value)
            num_clients_from_europe = num_list[0][0]
            self.send_metric(Metrics.NUMBER_OF_CLIENTS_FROM_EUROPE, num_clients_from_europe)
            time.sleep(15)

    # Function to collect the total number of active clients from Asia and send it as a
metric to Zabbix server
    def collect_total_clients_from_asia(self):
        start_time = datetime.datetime.now()
        end_time = start_time + datetime.timedelta(minutes=6)
        while datetime.datetime.now() < end_time:
            print("Monitoring asia...")
            num_list = get_login_clients_by_location(Location.ASIA.value)
```

```
            num_clients_from_asia = num_list[0][0]
            self.send_metric(Metrics.NUMBER_OF_CLIENTS_FROM_ASIA, num_clients_from_asia)
            time.sleep(15)

    # Function to collect the total number of active clients from North America and send it
as a metric to Zabbix server
    def collect_total_clients_from_north_america(self):
        start_time = datetime.datetime.now()
        end_time = start_time + datetime.timedelta(minutes=6)
        while datetime.datetime.now() < end_time:
            print("Monitoring north america...")
            num_list = get_login_clients_by_location(Location.NORTH_AMERICA.value)
            num_clients_from_north_america = num_list[0][0]
            self.send_metric(Metrics.NUMBER_OF_CLIENTS_FROM_NORTH_AMERICA,
num_clients_from_north_america)
            time.sleep(15)

# Create a list of monitoring methods to be executed
monitoring_methods = [
    ZabbixMonitoring().collect_total_clients,
    ZabbixMonitoring().collect_total_users,
    ZabbixMonitoring().collect_total_admins,
    ZabbixMonitoring().collect_total_clients_from_europe,
    ZabbixMonitoring().collect_total_clients_from_asia,
    ZabbixMonitoring().collect_total_clients_from_north_america
]
```

Додаток Е

Додаткові функції для застосунку.

```
import mysql.connector
from app.schema import UserLogin

# function to execute a database query and returns the result
def run_query(query, param, is_update):
    cnx = mysql.connector.connect(user='mysql', password='root', host='localhost',
port=3307, database='user')
    cursor = cnx.cursor()
    cursor.execute(query, param)
    if is_update:
        cnx.commit()
        return
    return cursor.fetchall()

# function to check if a user exists in the database
def check_user(data: UserLogin):
    query = """SELECT email, password FROM users
    where email=%s and password=%s"""
    val = (data.email, data.password)
    result = run_query(query, val, is_update=False)
    if not result:
        return False
    return True

# function to set the active status of a user in the database
def set_active(user: UserLogin, value: bool):
    query = """UPDATE users
       SET is_active = %s
       where email=%s and password=%s"""
```

```
val = (int(value), user.email, user.password)
run_query(query, val, is_update=True)
```