НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Навчально-науковий інститут атомної та теплової енергетики

Кафедра цифрових технологій в енергетиці

«До захисту допущено»
Завідувач кафедри
_____Наталія АУШЕВА
“___”_____2026 р.

Дипломна робота
на здобуття ступеня бакалавр

За освітньою програмою “Цифрові технології в енергетиці”

Спеціальності 122 “Комп’ютерні науки”

на тему: “Мережевий експортер метрик вузла у форматі OpenMetrics і його інтеграція в систему моніторингу”

Виконав: студент 4 курсу, групи ТР-21
Малюк Ілля Олександрович _____
(прізвище, ім’я, по батькові) (підпис)

Керівник доцент, к.т.н, доцент Лариса КУБЛІЙ _____
(посада, науковий ступінь, вчене звання, ім’я, ПРІЗВИЩЕ) (підпис)

Рецензент: доцент кафедри теплової та альтернативної
енергетики, к.т.н., доцент Олександр СІРИЙ _____
(посада, науковий ступінь, вчене звання, ім’я, ПРІЗВИЩЕ) (підпис)

Н.контроль ст.викладач Ольга БЕСПАЛА _____
(посада, ім’я, ПРІЗВИЩЕ) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ - 2026

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ ЕНЕРГЕТИКИ

Кафедра ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ

Рівень вищої освіти – перший (бакалаврський)
спеціальність 122 “Комп’ютерні науки”
Освітньо-професійна програма “Цифрові технології в енергетиці”

ЗАТВЕРДЖУЮ
Завідувач кафедри ЦТЕ
_____Наталія АУШЕВА
(підпис)
“___” _____2026 р.

ЗАВДАННЯ

на дипломну роботу студенту

_____ Малюку Іллі Олександровичу_____
(прізвище, ім’я, по батькові)

1. Тема роботи “Мережевий експортер метрик вузла у форматі OpenMetrics і його інтеграція в систему моніторингу”

Науковий керівник Кублій Лариса Іванівна, к.т.н, доцент
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від “28” травня 2026 року № 1992

2. Термін подання студентом роботи 04.06.2026

3. Вихідні дані до роботи мова програмування Go, мова C (для модулів eBPF), бібліотека React, стандарт OpenMetrics, операційна система Linux, середовище розробки GoLand

4. Перелік питань, які потрібно розробити _____

1) провести аналіз методів та алгоритмів збору мережевої статистики в ОС Linux;

2) проаналізувати наявні програмні рішення та обґрунтувати вибір технологій;
3) розробити архітектуру і програмний код мережевого експортера на базі eBPF;
4) створити графічний інтерфейс (дашборд) для візуалізації зібраних метрик;
5) провести контейнеризацію й тестування розробленої програмної системи.

5. Орієнтований перелік ілюстративного матеріалу структурна схема взаємодії віртуальної машини eBPF із ядром ОС Linux, алгоритм агрегації метрик транспортного рівня, структурна схема архітектури комплексу моніторингу, скріншоти розробленого графічного інтерфейсу (дашборду), схема CI/CD-конвеєра.

6. Дата видачі завдання 15.09.2025

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1.	Вибір теми роботи	15. 09. 2025	Виконано
2.	Аналіз методів та засобів розв’язання задачі	20.04-24.04.26	Виконано
3.	Розробка архітектури та загальної структури системи	24.04-27.04.26	Виконано
4.	Розробка окремих підсистем	28.04-01.05.26	Виконано
5.	Програмна реалізація системи	02.05-07.05.26	Виконано
6.	Оформлення пояснювальної записки	08.05-29.05.26	Виконано
7.	Захист програмного забезпечення	12.05.26	Виконано
8.	Передзахист	02.06.26	Виконано
9.	Захист	18.06.2026	Виконано

Студент _____ Малюк І. О.
(підпис) (прізвище та ініціали)

Керівник _____ Кублій Л. І.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Дипломна робота виконана на 61 сторінці, містить 22 ілюстрації, 1 таблицю, 2 додатки, 22 джерел у переліку посилань.

Мета роботи — створення ефективної програмної системи для глибокого моніторингу мережевої інфраструктури серверного вузла Linux із застосуванням інноваційних технологій перехоплення системних викликів та експортом телеметрії у стандарті OpenMetrics.

Методи й засоби: технологія eBPF (Extended Berkeley Packet Filter) для безпечного перехоплення подій у ядрі операційної системи; парсинг віртуальної файлової системи procfs; мови програмування Go (для реалізації бекенд-експортера) та C (для написання низькорівневих зондів ядра); бібліотека React і мова TypeScript для розробки клієнтського веб-дашборду; стандарт OpenMetrics; технології контейнеризації Docker.

Результати: розроблено повноцінний мережевий експортер, який забезпечує можливість відстежувати стан підсистеми Conntrack і деталізувати втрати пакетів і повторні передачі TCP до рівня конкретних IP-адрес без створення надмірного навантаження на систему (zero-overhead). Створено спеціалізований інтерактивний графічний інтерфейс для візуалізації багатовимірних мережевих метрик у режимі реального часу, виконано інтеграцію системи в середовище оркестрації за допомогою контейнеризації.

Ключові слова: МЕРЕЖЕВИЙ МОНІТОРИНГ, ЕКСПОРТЕР МЕТРИК, ЯДРО LINUX, EBPF, OPENMETRICS, GO, REACT, КОНТЕЙНЕРИЗАЦІЯ.

ABSTRACT

The diploma thesis consists of 61 pages, contains 22 illustrations, 1 tables, 2 appendices and 22 references.

The purpose of the work is to create an efficient software system for deep monitoring of the Linux server node network infrastructure using innovative system call interception technologies and exporting telemetry in the OpenMetrics standard.

Methods and tools: eBPF (Extended Berkeley Packet Filter) technology for safe interception of events in the operating system kernel; parsing of the procfs virtual file system; Go (for backend exporter implementation) and C (for writing low-level kernel probes) programming languages; React library and TypeScript language for developing the client web dashboard; OpenMetrics standard; Docker containerization technologies.

Results: a fully-fledged network exporter was developed, which allows monitoring the state of the Conntrack subsystem and detailing packet drops and TCP retransmissions down to the level of specific IP addresses without creating excessive system overhead (zero-overhead). A specialized interactive graphical interface was created to visualize multidimensional network metrics in real-time, and the system was successfully integrated into an orchestration environment via containerization.

Keywords: NETWORK MONITORING, METRICS EXPORTER, LINUX KERNEL, EBPF, OPENMETRICS, GO, REACT, CONTAINERIZATION.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

API — Application Programming Interface (інтерфейс програмування застосунків)

DOM — Document Object Model (об’єктна модель документа)

eBPF — Extended Berkeley Packet Filter (розширений пакетний фільтр Берклі) — технологія, яка дає можливість безпечно виконувати ізольовані програми у просторі ядра Linux без зміни його вихідного коду або завантаження модулів ядра.

HTTP — HyperText Transfer Protocol (протокол передачі гіпертексту)

ICMP — Internet Control Message Protocol (міжмережевий протокол керування повідомленнями)

IDE — Integrated Development Environment (інтегроване середовище розробки)

PromQL — Prometheus Query Language (мова запитів для системи моніторингу Prometheus)

Rx / Tx — Receive / Transmit (отримання / передача мережевих даних)

TCP/IP — Transmission Control Protocol / Internet Protocol (набір мережевих протоколів передачі даних)

TSDB — Time Series Database (база даних часових рядів) — система керування базами даних, оптимізована для зберігання та обслуговування масивів даних, прив’язаних до часу.

UDP — User Datagram Protocol (протокол передачі датаграм користувача)

Conntrack (Connection Tracking) — підсистема мережевого фільтра ядра Linux, що відповідає за відстеження стану активних мережевих з’єднань.

procfs — віртуальна файлова система (зазвичай монтується в каталозі /proc) в UNIX-подібних операційних системах, яка надає інформацію про поточні процеси та загальний стан ядра.

Скрапінг (Scraping) — у контексті систем моніторингу: процес періодичного автоматичного збору (витягування) метрик із цільових серверів або експортерів.

ВСТУП

У сучасній індустрії інформаційних технологій домінуючою парадигмою розгортання програмного забезпечення є використання хмарно-орієнтованих (cloud-native) архітектур, контейнеризації та мікросервісного підходу. У таких розподілених системах мережева інфраструктура відіграє критичну роль, оскільки від її стабільності й пропускної здатності залежить працездатність усього програмного комплексу. Разом із тим, зростання складності мережевих взаємодій призводить до виникнення неочевидних аномалій, таких як мікро-сплески трафіка (microbursts), вичерпання лімітів таблиць маршрутизації або втрата пакетів на рівні транспортних протоколів.

Традиційні підходи до системного моніторингу, які базуються на опитуванні стандартних лічильників операційної системи (наприклад, через SNMP або базовий парсинг утиліт системного адміністрування), не здатні забезпечити достатній рівень спостережуваності (observability). Вони надають тільки агреговану статистику, не дозволяючи деталізувати проблеми до рівня конкретних IP-адрес чи мережевих з’єднань. З іншого боку, використання інструментів глибокого захоплення пакетів генерує надмірне навантаження на центральний процесор і є неприйнятним для постійного моніторингу високонавантажених серверів. Відповідно, задача розробки спеціалізованого, високопродуктивного програмного забезпечення, здатного здійснювати глибокий моніторинг мережевого стека ядра Linux із мінімальними накладними витратами (zero-overhead), є вкрай актуальною.

Мета роботи — створення програмної системи глибокого мережевого моніторингу інфраструктури вузла із застосуванням технології eBPF та експортом телеметричних даних у форматі OpenMetrics.

Для досягнення поставленої мети необхідно розв’язати такі завдання:

— проаналізувати підходи, методи й алгоритми розв’язання задачі збору та агрегації статистики функціонування мережевого стека ОС Linux;

— проаналізувати програмні засоби для реалізації програмної системи збору, експорту й візуалізації мережевих метрик;

— розробити програмне забезпечення мережевого експортера на базі технології eBPF та інтерактивного графічного дашборду;

— виконати тестування розробленого програмного забезпечення в умовах імітації мережевих аномалій і навантажень.

Об’єкт розробки — процес збору та аналізу телеметричних даних про стан мережевої підсистеми операційної системи.

Предмет розробки — методи та інструментальні засоби низькорівневого перехоплення подій мережевого стека, алгоритми розрахунку телеметричних показників та їхньої подальшої візуалізації.

Апробація результатів роботи — основні положення, архітектурні рішення і практичні результати розробки доповідалися та отримали позитивну оцінку на XXIII Міжнародній науково-практичній конференції молодих вчених та студентів «Сучасні проблеми наукового забезпечення енергетики: безпека, стійкість, ІТ та екологічний моніторинг» (м. Київ, 17-22 квітня 2026 р) [1].

Структура й обсяг роботи — дипломна записка складається зі вступу, чотирьох розділів, висновків, списку використаних джерел із 22 найменувань та 2 додатків. Загальний обсяг роботи становить 61 сторінку, в тому числі 22 рисунки і 1 таблиця.

1. АНАЛІЗ ПРОБЛЕМАТИКИ І ПОСТАНОВКА ЗАДАЧІ СТВОРЕННЯ МЕРЕЖЕВОЇ СИСТЕМИ МОНІТОРИНГУ

У розділі розглядається проблематика забезпечення безперервної спостережуваності в розподілених інформаційних системах. Проведено аналіз функціональних і нефункціональних вимог до програмного забезпечення, яке розробляється. Виконано огляд наявних методів отримання системної статистики, а також порівняльний аналіз сучасних програмних засобів моніторингу для обґрунтування доцільності створення нового спеціалізованого рішення.

1.1. Прикладна задача моніторингу мережевого стека

Сучасні серверні платформи на базі ядра Linux обробляють мільйони мережевих пакетів за секунду. В умовах експлуатації мікросервісних застосунків або систем балансування навантаження виникає необхідність безперервного контролю за станом транспортного рівня (модель OSI), зокрема протоколів TCP і UDP [2]. Прикладна задача, яка розв’язується в межах дипломної роботи, полягає у створенні комплексного інструментарію, який дасть можливість автоматизувати процес виявлення прихованих мережевих деградацій.

Першочерговою підзадачею є відстеження втрат пакетів (packet drops) і повторних передач (TCP retransmissions). У стандартних умовах ядро операційної системи тихо відкидає пакети при переповненні апаратних буферів або порушенні правил маршрутизації, інкрементуючи лише глобальні системні лічильники. Відсутність контексту (адреси відправника та отримувача) унеможливлює швидку локалізацію проблеми. Тому розроблювана система повинна розв’язати задачу видобування контекстної інформації безпосередньо з пам’яті ядра під час виникнення інциденту.

Наступною важливою підзадачею є моніторинг підсистеми відстеження з’єднань (Connection Tracking). Механізм nf_conntrack відповідає за збереження

стану всіх активних логічних з’єднань. У випадку досягнення ліміту записів у цій таблиці (наприклад, під час DDoS-атак типу SYN-flood), сервер припиняє обробку нових запитів. Відповідно, програмне забезпечення має превентивно відстежувати рівень заповненості цієї таблиці та експортувати відповідну метрику.

Крім розширеної аналітики, система повинна розв’язувати задачу інтеграції з існуючим індустріальним стеком. Дані мають передаватися у стандартизованому форматі OpenMetrics, що дозволить використовувати часову базу даних як надійне сховище. Окремою прикладною задачею є створення графічного інтерфейсу користувача (дашборду), який дозволить системним адміністраторам та інженерам з надійності (DevOps/SRE) інтуїтивно зрозуміло інтерпретувати зібрані багатовимірні дані в режимі реального часу.

З архітектурної точки зору, система повинна відповідати жорстким нефункціональним вимогам щодо споживання ресурсів: процес збору метрик не повинен викликати блокувань у ядрі або суттєво конкурувати за процесорний час із цільовими бізнес-застосунками, які виконуються на вузлі.

Загальна схема взаємодії користувачів та зовнішніх систем із розроблюваним програмним комплексом у вигляді діаграми прецедентів (Use Case) наведена на рисунку 1.1.

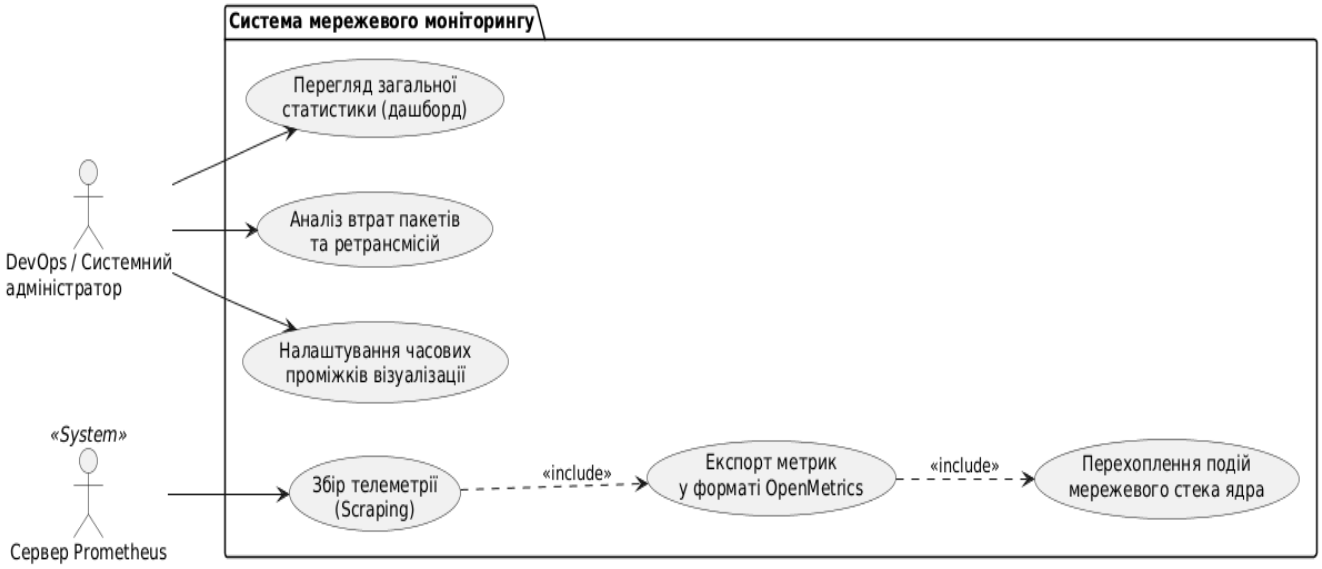


Рисунок 1.1 — Діаграма прецедентів системи мережевого моніторингу

Діаграма прецедентів чітко окреслює функціональні межі розроблюваного комплексу й визначає основні сценарії використання. Для практичної реалізації зазначених прецедентів необхідно ретельно дослідити наявні програмні механізми взаємодії з ядром операційної системи, які детально розглядаються далі.

1.2. Аналіз методів та підходів до збору системної мережевої статистики

Для реалізації ефективної системи моніторингу необхідно визначити найбільш оптимальні методи взаємодії з ядром операційної системи Linux для отримання метрик. Історично склалося кілька фундаментальних підходів до вилучення телеметричної інформації з мережевого стека, кожен з яких має свої переваги і суттєві обмеження в контексті побудови високонавантажених хмарних середовищ.

Перший, найбільш традиційний підхід базується на використанні віртуальної файлової системи procfs. Ця підсистема діє як інтерфейс до внутрішніх структур даних ядра. Програми простору користувача (user space) можуть отримувати статистику щодо кількості переданих байтів, пакетів і базових помилок шляхом звичайного читання псевдофайлів, таких як /proc/net/dev або /proc/net/snmp. Перевагою цього підходу є простота реалізації і відсутність необхідності встановлення додаткових модулів ядра. Проте недоліком є високий ступінь агрегації даних. Віртуальна файлова система надає лише загальні лічильники (counters) для інтерфейсів або протоколів у цілому, що унеможливлює деталізацію статистики до рівня конкретних мережевих з’єднань або причин виникнення відмов.

Другий підхід передбачає використання спеціалізованих мережевих сокетів Netlink. Механізм Netlink використовується для двобічного зв’язку між ядром і програмами користувача. Він дає можливість отримувати інформацію про маршрутизацію, стан мережевих інтерфейсів і детальну статистику TCP-з’єднань (через підсистему sock_diag). Цей метод значно ефективніший за парсинг текстових файлів procfs, оскільки дані передаються в бінарному форматі, що знижує навантаження на процесор під час серіалізації / десеріалізації. Проте Netlink

все ще оперує концепцією опитування (polling), що означає необхідність періодичного надсилання запитів до ядра для отримання актуального зрізу стану.

Третій підхід — глибока інспекція пакетів (Deep Packet Inspection) за допомогою таких бібліотек, як libpcap [3]. Цей метод дає можливість перехоплювати копію кожного мережевого пакета, який проходить через мережеву карту, і аналізувати його вміст. Незважаючи на абсолютну повноту інформації, цей підхід супроводжується катастрофічними накладними витратами. Копіювання кожного пакета з простору ядра в простір користувача призводить до експоненційного зростання споживання процесорного часу й переповнення системної шини пам'яті, що робить його абсолютно непридатним для безперервного моніторингу на швидкостях 10 Gbps і вище.

Останній, найбільш інноваційний підхід, який було обрано в якості базового для розробки мережевого експортера, базується на технології eBPF (Extended Berkeley Packet Filter). Технологія eBPF дає можливість динамічно завантажувати та безпечно виконувати спеціально скомпільовані мікропрограми (зонди) безпосередньо в ядрі операційної системи. Замість того, щоб копіювати пакети або періодично опитувати ядро, eBPF реалізує подійно-орієнтовану модель (event-driven). Зонди прикріплюються до конкретних функцій ядра (наприклад, функцій скидання пакетів kfree_skb або повторної передачі tcp_retransmit_skb). При виклику такої функції зонд миттєво збирає необхідний контекст (IP-адреси, порти) і асинхронно передає його програмі-експортеру в простір користувача через спеціальні структури даних — eBPF Maps. Аналіз показав, що такий підхід забезпечує найвищу точність даних при мінімально можливому споживанні обчислювальних ресурсів.

1.3. Порівняльний аналіз наявних програмних засобів системного і мережевого моніторингу

Для обґрунтування доцільності розробки власного програмного забезпечення необхідно провести глибокий аналіз наявних на ринку індустріальних рішень.

Сучасний набір інструментів спостережуваності (observability) пропонує широкий спектр систем, проте більшість із них є універсальними і не здатні ефективно ирозв’язувати вузькоспеціалізовані задачі глибокого мережевого трасування без суттєвих накладних витрат. Найбільш поширеними агентами моніторингу є Zabbix Agent [4], Telegraf [5], Datadog Agent [6] і Prometheus Node Exporter [7].

Програмний засіб Zabbix є однією з найстаріших і найпоширеніших систем моніторингу інфраструктури (Enterprise-рівня). Агент Zabbix, який встановлюється на цільові вузли, підтримує збір базової мережевої статистики через системні виклики й читання віртуальних файлових систем. Перевагою системи є її зрілість, наявність вбудованої системи тригерів, підтримка практично всіх операційних систем.

Проте в умовах сучасних хмарних (cloud-native) інфраструктур архітектура Zabbix виявляє суттєві недоліки. Зберігання метрик у реляційних базах даних (наприклад, PostgreSQL або MySQL) робить систему вразливою до високих навантажень при збереженні часових рядів із високою частотою. З точки зору мережевого моніторингу агент Zabbix не підтримує технологію eBPF з коробки. Для отримання деталізованої статистики (наприклад, TCP-ретрансмісій для конкретних IP-адрес) необхідно писати кастомні UserParameter-скрипти, які викликатимуть сторонні утиліти (наприклад, tshark або tcpdump), що призведе до катастрофічного споживання ресурсів центрального процесора.

Інструмент Telegraf (InfluxData) — це агент для збору й надсилання метрик, написаний мовою Go, який є частиною екосистеми TICK-стека. Основною перевагою Telegraf є його плагінна архітектура: система підтримує понад 200 вхідних плагінів (Input Plugins).

Для мережевого моніторингу використовуються плагіни net і netstat, які здійснюють парсинг псевдофайлів каталогу /proc. Засіб Telegraf забезпечує високу швидкість обробки даних і підтримку формату Prometheus. Проте, при необхідності відстеження подій на рівні ядра (наприклад, стану таблиці nf_conntrack або мікро-сплесків втрат пакетів), універсальність агента стає його слабким місцем. Наявні плагіни для роботи з eBPF мають експериментальний статус, є складними у

конфігурації і не дають можливості динамічно прикріплювати зонди до специфічних функцій ядра, таких як tcp_retransmit_skb, без написання власних модулів мовою C.

Система Datadog є комерційною SaaS-платформою спостережуваності. Офіційний Datadog Agent включає модуль Network Performance Monitoring (NPM), який архітектурно є найбільш близьким до об’єкта розробки дипломної роботи, оскільки він активно використовує eBPF для інспекції мережевого стека Linux.

Цей агент здатний малювати детальну топологію мережі та відстежувати затримки на рівні окремих процесів. Недоліком цього рішення є його пропрієтарність. Використання Datadog прив’язує інфраструктуру до одного постачальника (vendor lock-in) і вимагає значних фінансових витрат при масштабуванні кластера. Крім того, закритий вихідний код агента унеможливлює його аудит та адаптацію (кастомізацію) під специфічні внутрішні вимоги безпеки підприємства.

Експортер Prometheus Node Exporter є стандартом де-факто для експозиції апаратних і системних метрик Linux-вузлів у середовищах Kubernetes. Він написаний мовою Go, є надзвичайно легковаговим і повністю відповідає ідеології OpenMetrics.

Мережевий колектор Node Exporter збирає дані про кількість отриманих і переданих байтів (node_network_receive_bytes_total), пакетів і базових помилок для кожного інтерфейсу. Проте, згідно з офіційною філософією розробників Prometheus, Node Exporter принципово не збирає метрики високої кардинальності (high cardinality). Тобто, він не відстежує стан окремих мережевих з’єднань, стан таблиць conntrack або адреси, з якими відбуваються втрати пакетів, оскільки це може призвести до експоненційного зростання обсягу даних.

Виходячи з наведеного аналізу, можна зробити висновок, що наявні відкриті рішення або надають лише поверхневу агреговану статистику, або вимагають надмірних обчислювальних ресурсів, у той час як потужні eBPF-рішення є комерційними й закритими. Це формує обґрунтовану потребу в розробці власного відкритого мережевого експортера.

Для узагальнення проведеного дослідження результати порівняння архітектурних особливостей і функціональних можливостей розглянутих систем із розробленим програмним забезпеченням зведено до таблиці 1.1.

Таблиця 1.1 — Порівняльний аналіз програмних засобів мережевого моніторингу

Критерій порівняння	Zabbix Agent	Telegraf	Datadog Agent	Node Exporter	Розроблений експортер
Архітектура моніторингу	Pull / Push	Push	Push (до SaaS)	Pull	Pull
Мова розробки агента	C	Go	Go / Python	Go	Go / C
Глибока інспекція (eBPF)	Ні	Експериментально	Так	Ні	Так
Споживання ресурсів CPU	Середнє	Середнє	Високе	Низьке	Низьке (Zero-overhead)
Модель поширення	Open Source	Open Source	Пропрієтарна	Open Source	Open Source
Формат експорту даних	Власний формат	Мультиформатний	Власний (SaaS)	OpenMetrics	OpenMetrics
Відстеження TCP ретрансмісій (per-IP)	Ні	Ні	Так	Ні	Так
Моніторинг nf_conntrack	Частково (через скрипти)	Так (окремий плагін)	Так	Частково	Так (з деталізацією)

Аналіз даних таблиці 1.1 доводить, що розроблене програмне забезпечення поєднує в собі переваги відкритих легковагових експортерів (сумісність з Prometheus, низьке споживання ресурсів, формат OpenMetrics) із розширеними можливостями комерційних агентів (глибока пакетна інспекція за допомогою eBPF, деталізація транспортного рівня). Це дає можливість стверджувати, що обраний технологічний стек є найбільш оптимальним для розв’язання поставленої задачі.

2. МЕТОДИ Й АЛГОРИТМИ ЗБОРУ МЕРЕЖЕВОЇ ТЕЛЕМЕТРІЇ В ОС LINUX

У розділі наведено детальне теоретичне обґрунтування та опис методів і алгоритмів, які лягли в основу розробленого програмного забезпечення. Розглянуто архітектуру мережевого стека операційної системи Linux, механізми роботи підсистеми відстеження з'єднань (Connection Tracking) та віртуальної файлової системи procfs. Особливу увагу приділено алгоритмічним основам технології eBPF (Extended Berkeley Packet Filter) як основного інструменту для низькорівневої, високоефективної інспекції мережевих пакетів та ідентифікації втрат на транспортному рівні.

2.1. Теоретичні основи функціонування мережевого стека Linux і підсистеми Conntrack

Мережевий стек ядра Linux — це складна, багаторівнева програмна абстракція, яка реалізує набір протоколів TCP/IP відповідно до еталонної моделі взаємодії відкритих систем (OSI). Процес обробки мережевого трафіка всередині ядра поділяється на етапи інкапсуляції (при відправленні) та декапсуляції (при отриманні) пакетів.

Коли мережевий інтерфейс (Network Interface Card, NIC) отримує кадр (frame) з фізичного середовища, він ініціює апаратне переривання. Драйвер пристрою обробляє це переривання, виділяє структуру даних sk_buff (socket buffer) у пам'яті ядра і копіює туди вміст кадру. Структура sk_buff є фундаментальною структурою даних у мережевій підсистемі Linux, через яку пакет передається між усіма рівнями мережевого стека без необхідності повного копіювання його вмісту. Далі пакет передається на рівень протоколу IP (мережевий рівень), де виконується перевірка контрольних сум, обробка таблиць маршрутизації (FIB — Forwarding Information Base) і правил мережевого екрана (Netfilter) [8].

Критично важливою складовою фреймворку Netfilter, яка потребує постійного моніторингу, є підсистема nf_conntrack (Connection Tracking). Цей модуль відповідає за відстеження стану всіх логічних мережевих з'єднань, що проходять через сервер.

Алгоритм роботи nf_conntrack базується на веденні глобальної хеш-таблиці у просторі ядра. Коли надходить новий пакет (наприклад, TCP SYN), система аналізує його заголовки і створює новий запис (кортеж), який містить п'ять ключових ідентифікаторів: IP-адресу джерела, IP-адресу призначення, порт джерела, порт призначення, протокол рівня 4 (TCP/UDP). Цьому запису присвоюється стан NEW. Коли система фіксує двосторонній обмін пакетами, стан з'єднання переходить у ESTABLISHED. Якщо з'єднання закривається, воно переходить у стан TIME_WAIT і зберігається в таблиці ще деякий час для обробки запізнілих пакетів.

Основна проблема експлуатації цієї підсистеми полягає в тому, що розмір таблиці conntrack жорстко обмежений параметром nf_conntrack_max, оскільки кожен запис споживає несторінкову пам'ять ядра (Slab cache). Якщо швидкість надходження нових пакетів перевищує швидкість їхньої обробки або якщо на сервер здійснюється атака на відмову в обслуговуванні, кільцевий буфер заповнюється на 100%. Згідно з алгоритмом роботи ядра, за умови досягнення цього ліміту, всі наступні пакети, які не належать до вже встановлених з'єднань, безшумно відкидаються (packet drop) з повідомленням «nf_conntrack: table full, dropping packet».

З огляду на вищезазначене, алгоритм моніторингу цієї підсистеми в розроблюваному експортері полягає в безперервному розрахункові коефіцієнта заповненості таблиці K_c (рисунок 2.1), який визначається як відношення поточної кількості записів до максимально допустимої:

$$K_c = \frac{C_{current}}{C_{max}} \cdot 100\% \ ,$$

де K_c — поточна кількість активних з'єднань у таблиці відстеження; C_{max} — граничне значення розміру таблиці, встановлене конфігурацією ядра.

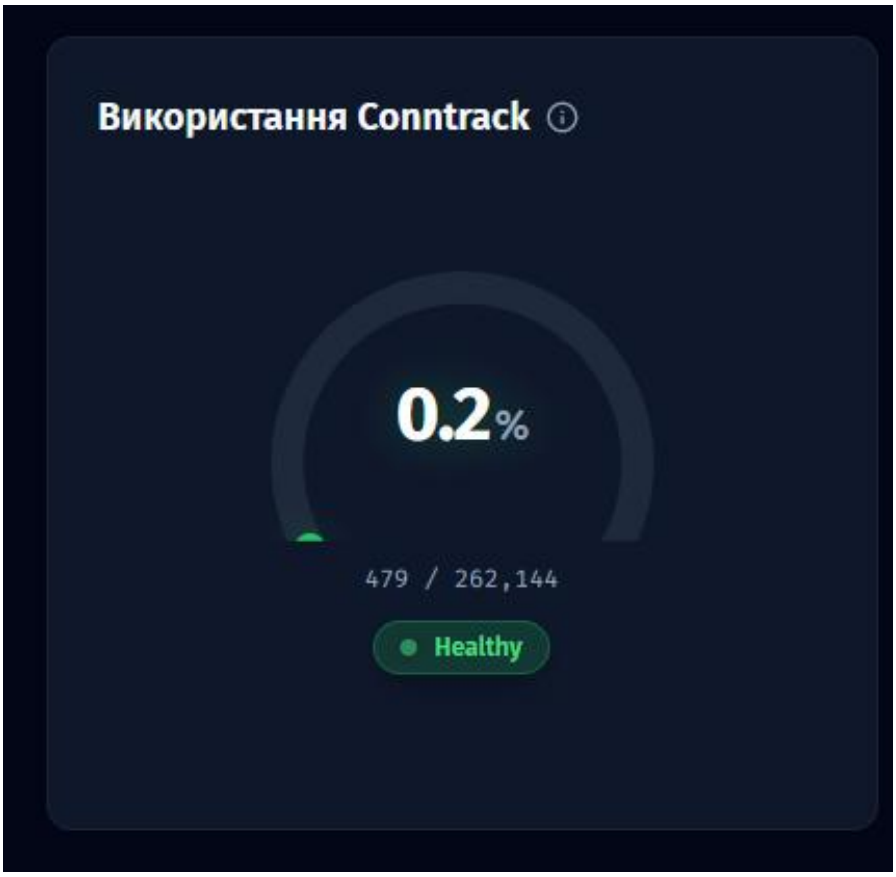


Рисунок 2.1 — Візуалізація коефіцієнта заповненості таблиці відстеження з’єднань

Таким чином, безперервний розрахунок і контроль цього коефіцієнта дає системі моніторингу змогу превентивно виявляти мережеві перевантаження ще до моменту фактичної відмови в обслуговуванні клієнтів.

2.2. Алгоритм збору базової мережевої статистики через віртуальну файлову систему procfs

Для отримання показників продуктивності мережевих інтерфейсів (швидкість передачі даних) і загальної статистики TCP/IP-стека в системі використано метод синтаксичного аналізу віртуальної файлової системи procfs. Ця файлова система, змонтована в каталозі /proc, не займає дискового простору,

функціонує як вікно до внутрішніх структур даних ядра, які генеруються «на льоту» під час запиту на читання.

Алгоритм збору даних із підсистеми procfs розроблено з урахуванням мінімізації споживання системних ресурсів. Він передбачає такі етапи:

— ініціалізація та відкриття файлових дескрипторів. Програма-експортер виконує системний виклик open() для файлів /proc/net/dev (статистика інтерфейсів), /proc/net/snmp (статистика протоколів) і /proc/net/sockstat (статистика сокетів);

— буферизоване читання (Buffered Reading). Зважаючи на те, що розмір псевдофайлів може бути значним (особливо за наявності десятків віртуальних інтерфейсів у контейнеризованих середовищах), алгоритм використовує буферизований сканер. Замість зчитування всього файлу в оперативну пам’ять (що призводить до зайвих алокацій), файл зчитується порядково. Це знижує навантаження на систему збору сміття (Garbage Collector) мови Go;

— лексичний аналіз і токенізація. Зчитаний рядок пропускається через функцію-токенізатор, яка ігнорує заголовки файлу і розділяє текстовий потік на масив підрядків (токенів), використовуючи пробіли або символи двокрапки як розділювачі;

— конвертація типів. Текстові репрезентації числових значень (наприклад, кількість прийнятих байтів «1056734») конвертуються в 64-бітні цілі числа без знака (uint64). Використання 64-бітних типів є обов’язковим, оскільки на серверах з інтерфейсами 10/100 Gbps 32-бітні лічильники переповнюються (integer overflow) за кілька секунд;

— агрегація і збереження в пам’яті. Декодовані значення групуються в спеціалізовані структури даних і готуються до трансформації у формат OpenMetrics.

Так, наприклад, при обробці файлу /proc/net/dev алгоритм ідентифікує кожний мережевий інтерфейс (наприклад, eth0 або docker0) і витягує з нього чотири ключові лічильники: Receive Bytes (отримані байти), Receive Packets (отримані пакети), Transmit Bytes (відправлені байти), Transmit Packets (відправлені пакети).

Для обчислення безпосередньої швидкості передачі даних (пропускної здатності) алгоритмічний апарат розробленої системи не виконує розрахунків на боці агента. Експортер передає монотонно зростаючі лічильники типу Counter до централізованої бази даних Prometheus. Розрахунок швидкості $V(t_i)$ у байтах за секунду виконується на етапі запиту даних системою візуалізації за допомогою диференціювання за часом:

$$V(t_i) = \frac{B(t_i) - B(t_{i-1})}{t_i - t_{i-1}} \quad ,$$

де $B(t_i)$ — значення лічильника байтів у поточний момент часу t_i; $B(t_{i-1})$ — значення лічильника байтів у попередній момент часу t_{i-1}.

Цей підхід, який делегує математичні обчислення часовій базі даних, повністю відповідає філософії мікросервісного моніторингу і дає можливість утримувати споживання процесорного часу експортером на рівні, близькому до нуля.

2.3. Алгоритмічні основи технології eBPF і механізм перехоплення подій ядра

Незважаючи на високу швидкість доступу до віртуальної файлової системи procfs, цей метод оперує виключно агрегованими лічильниками, що унеможливлює виконання глибокої пакетної інспекції і встановлення першопричин мережевих аномалій. Для розв’язання задачі отримання контекстно-залежної інформації безпосередньо з простору ядра (Kernel Space) у розробленій системі застосовано метод динамічного трасування на базі технології eBPF (Extended Berkeley Packet Filter).

Технологія eBPF [9] — високопродуктивна реєстрова віртуальна машина (Virtual Machine), інтегрована безпосередньо в ядро операційної системи Linux [10]. Вона дає можливість виконувати користувацькі мікропрограми (зонди) в ізольованому середовищі (sandbox) як реакцію на певні системні події, не

вимагаючи при цьому модифікації вихідного коду ядра чи завантаження нестабільних модулів ядра (Kernel Modules).

Алгоритм використання eBPF у розроблюваному мережевому експортері складається з двох фундаментальних фаз — фази підготовки в просторі користувача (User Space) і фази виконання в просторі ядра (рисунок 2.2).

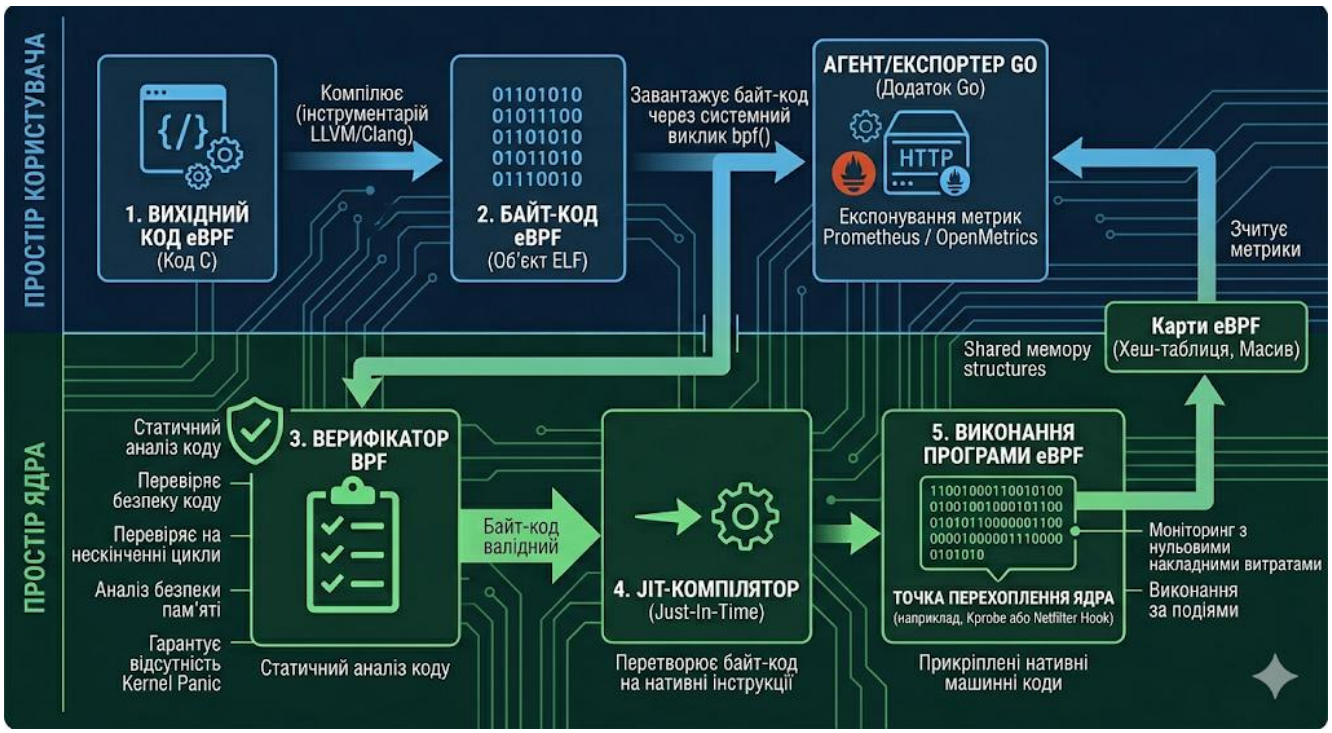


Рисунок 2.2 — Структурна схема взаємодії віртуальної машини eBPF з ядром ОС Linux

Взаємодія віртуальної машини eBPF з ядром ОС Linux відбувається в такій послідовності:

— компіляція. Програма-зонд, написана обмеженою підмножиною мови C (без використання стандартної бібліотеки libc і глобальних змінних), компілюється за допомогою інструментарію LLVM/Clang у специфічний байт-код формату ELF;

— завантаження і верифікація. Керуюча програма-агент (написана мовою Go) здійснює системний виклик bpf() для завантаження байт-коду в ядро. На цьому етапі спрацьовує критично важливий компонент — верифікатор BPF Verifier.

Верифікатор виконує статичний аналіз коду, будує граф потоку керування (Control Flow Graph), математично гарантує, що програма не містить нескінченних циклів, не виконує розіменування нульових вказівників (null pointer dereference) і має обмежену кількість інструкцій. Це гарантує абсолютну безпеку цільового сервера — зонд eBPF концептуально не здатний викликати збій ядра (Kernel Panic);

— JIT-компіляція. Після успішної верифікації байт-код транслюється JIT-компілятором (Just-In-Time) ядра в нативні машинні інструкції архітектури x86_64 або ARM64, що забезпечує швидкість виконання, порівнянну з нативним кодом ядра;

— прикріплення (Attachment). Скомпільована програма динамічно прикріплюється до точок перехоплення (hook points). У дипломній роботі використовуються динамічні зонди kprobes (Kernel Probes), які забезпечують можливість вставити обробник на вхід у будь-яку функцію ядра.

2.4. Метод ідентифікації мережевих аномалій на транспортному рівні

Основною цінністю технології eBPF для розроблюваного експортера є можливість ідентифікації конкретних вузлів-ініціаторів, з якими спостерігається деградація зв’язку. Розглянемо розроблений алгоритм перехоплення на прикладі відстеження повторних передач сегментів (TCP Retransmissions).

Повторна передача виникає тоді, коли відправник не отримує підтвердження (ACK) від отримувача протягом заданого тайм-ауту (RTO — Retransmission Timeout). У ядрі Linux за виконання цієї операції відповідає функція tcp_retransmit_skb. Відповідно, програма-зонд eBPF прикріплюється до цієї функції за допомогою kprobe.

Алгоритм обробки події всередині зонда передбачає таку послідовність дій:

— під час виклику tcp_retransmit_skb ядро передає функції вказівник на структуру sock (мережевий сокет), яка належить поточному з’єднанню;

— зонд eBPF перехоплює цей вказівник і здійснює безпечне читання пам’яті ядра за допомогою допоміжної функції bpf_probe_read_kernel();

— зі структури sock вилучаються ідентифікатори транспортного і мережевого рівнів: родина протоколів (IPv4 або IPv6), IP-адреса призначення (DADDR) та IP-адреса джерела (SADDR) ;

— вилучена IP-адреса використовується як ключ для хеш-функції.

Оскільки віртуальна машина eBPF повністю ізольована, вона не має прямого доступу до простору користувача. Для передачі зібраної статистики експортеру використовується спеціальний механізм спільної пам’яті — eBPF Maps (рисунок 2.3). У розробленій системі застосовано тип карти BPF_MAP_TYPE_RINGBUF (кільцевий буфер).

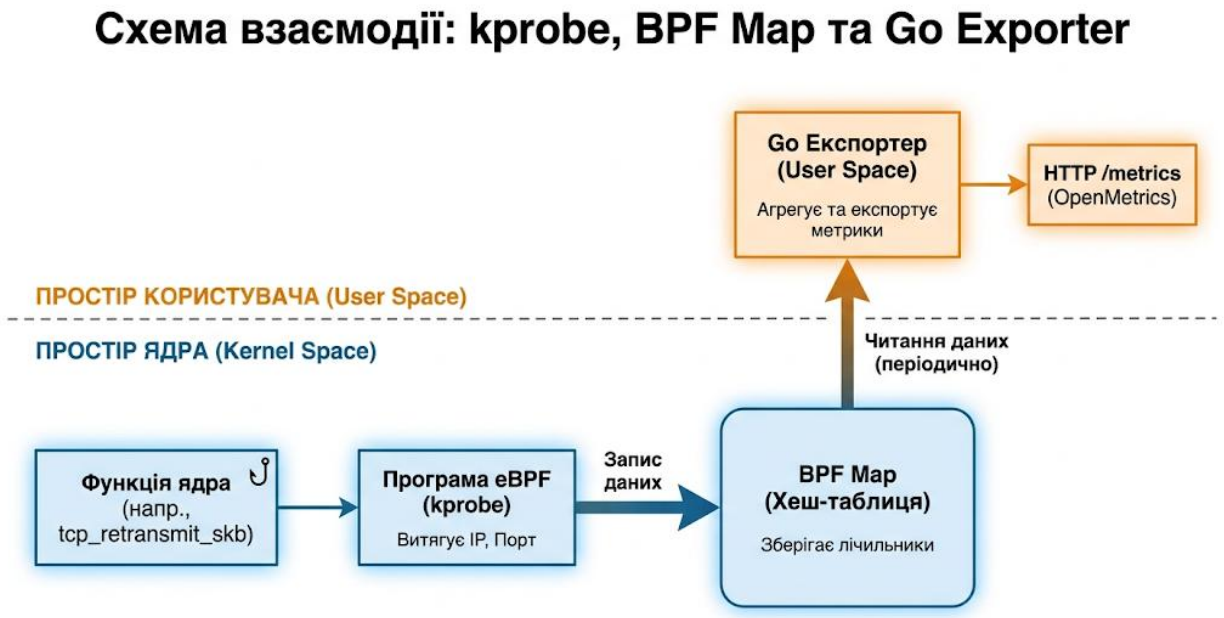


Рисунок 2.3 — Алгоритм агрегації метрик транспортного рівня через структуру BPF Map

Алгоритм комунікації такий:

— зонд в ядрі при перехопленні події виділяє блок пам’яті в кільцевому буфері (Ring Buffer) за допомогою функції bpf_ringbuf_reserve(). У цей блок

записується зібрана структура даних (IP-адреси, порти, мітка часу), після чого блок відправляється в простір користувача викликом bpf_ringbuf_submit();

— керуюча програма-експортер, написана мовою Go, яка працює в просторі користувача, створює спеціального слухача (Reader), який безперервно очікує на появу нових подій у буфері. При надходженні події агент миттєво зчитує сирі байти, десеріалізує їх у Go-структуру та оновлює відповідні лічильники Prometheus у своїй локальній пам’яті.

Аналогічний алгоритм застосовується для відстеження відкинутих пакетів (packet drops). У цьому випадку зонд прикріплюється до ядерної функції kfree_skb, яка викликається ядром для звільнення пам’яті мережевого буфера sk_buff у разі неможливості подальшої обробки пакета (наприклад, через хибні контрольні суми або відсутність маршруту). Зонд аналізує причину відкидання і вилучає IP-адресу з відповідних заголовків.

Завдяки реалізації подійно-орієнтованого алгоритму, система уникає накладних витрат на контекстне перемикання (Context Switch). Дані не копіюються між просторами ядра і користувача безперервно; передача відбувається тільки у вигляді суворо агрегованих числових значень раз на секунду. Це дає змогу розробленому мережевому експортеру функціонувати в умовах високоінтенсивного трафіка без впливу на показники MTTR (Mean Time To Recovery) та latency цільових застосунків.

3. ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ МЕРЕЖЕВОГО МОНІТОРИНГУ

У розділі описано практичний етап розробки програмного комплексу мережевого експортера та інтерактивної панелі візуалізації. Наведено детальну архітектуру системи, схеми взаємодії між простором ядра і простором користувача, а також розглянуто структурну організацію програмного коду серверної частини, написаної мовою Go, і клієнтського вебзастосунку. Окрему увагу приділено механізмам конкурентної обробки даних і забезпеченню безпеки під час виконання низькорівневих операцій трасування

3.1. Архітектурна структура програмного комплексу і потоки даних

Розроблене програмне забезпечення спроєктовано за децентралізованою мікросервісною архітектурою, що дає можливість мінімізувати взаємну залежність компонентів (loose coupling) і забезпечити високу стійкість системи до відмов. Програмний комплекс функціонально поділений на три автономні модулі: низькорівневий модуль збору даних (eBPF Kernel Probes), сервісний агент-агрегатор (Backend Exporter мовою Go), клієнтський веб-інтерфейс (Frontend Dashboard на базі React).

Організація потоків даних у системі підпорядкована подійно-орієнтованій та асинхронній моделям обробки інформації. Архітектурний життєвий цикл телеметрії реалізується за такою схемою взаємодії компонентів (рисунок 3.1):

— рівень ядра (Kernel Space). Мікропрограми eBPF, інтегровані в критичні точки мережевого стека, реагують на апаратні й програмні події (виклики функцій tcp_retransmit_skb і kfree_skb). Вони виконують первинну фільтрацію та інкрементують лічильники у високопродуктивних структурах спільної пам’яті (eBPF Maps), які функціонують як локальні сховища ключ-значення;

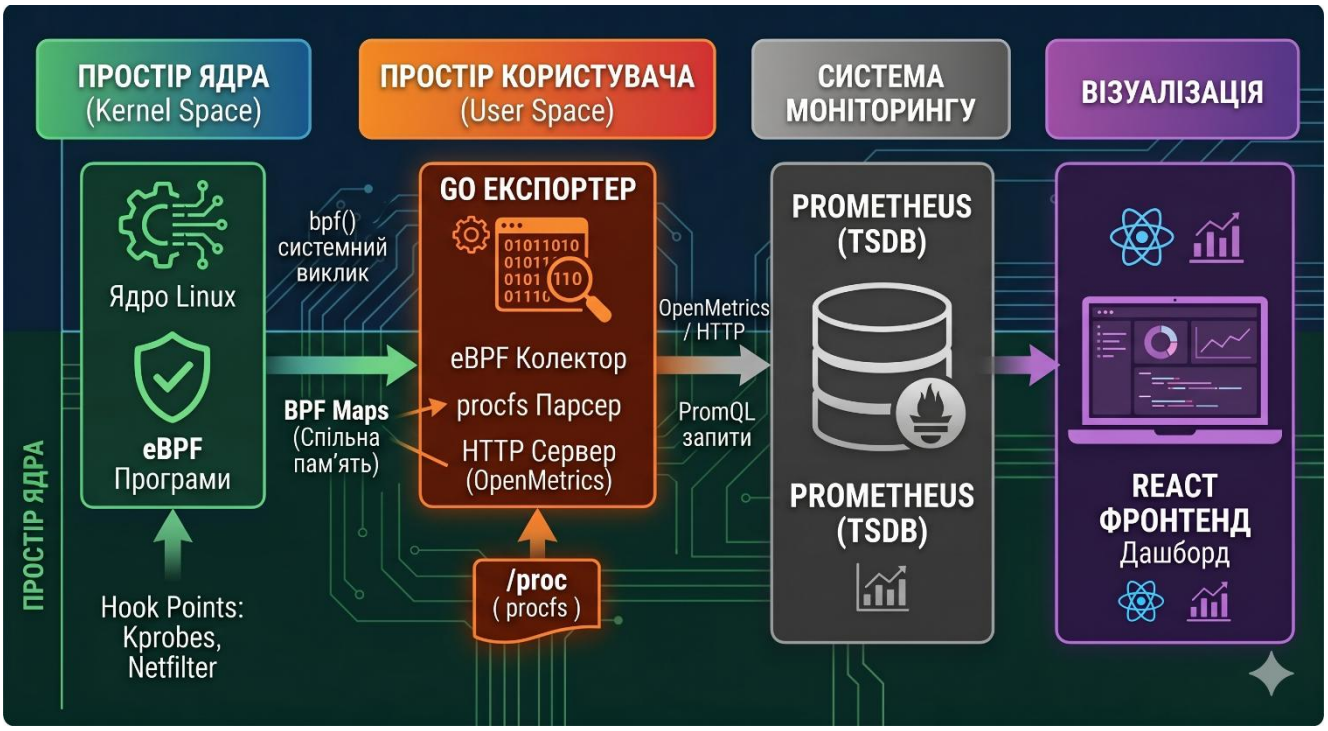


Рисунок 3.1 — Структурна схема архітектури комплексу моніторингу і напрямки потоків даних

— рівень сервісного агента (User Space Backend). Фоновий процес (демон), написаний мовою Go, функціонує з найвищими системними привілеями для доступу до підсистеми bpf(). Агент містить внутрішній таймер, який з інтервалом в одну секунду ініціює зчитування даних із eBPF Maps та паралельно виконує порядковий парсинг псевдофайлів віртуальної файлової системи procfs. Отримані сирі дані проходять процес валідації, нормування і структурування у внутрішніх об'єктах пам'яті застосунку;

— рівень експозиції (Metrics Exposure). Усередині Go-агента функціонує легковаговий HTTP-сервер. При надходженні вхідного запиту за HTTP-маршрутом /metrics від центрального сховища Prometheus, бекенд динамічно перетворює структури даних із внутрішньої пам'яті в лінійний текстовий формат OpenMetrics і повертає його в тілі HTTP-відповіді;

— рівень збереження й візуалізації. База даних часових рядів Prometheus [11] виконує періодичний скрапінг ендпоінту експортера. Клієнтський веб-застосунок (React Dashboard), який виконується в браузері користувача, взаємодіє з Prometheus

за допомогою HTTP API, надсилаючи аналітичні запити мовою PromQL та отримуючи масиви точок даних для побудови інтерактивних графіків.

Запропонована архітектурна схема забезпечує надійний, асинхронний потік телеметричних даних від ізольованого простору ядра до кінцевого користувача, гарантуючи при цьому високу швидкість обробки інформації.

3.2. Програмна реалізація серверного компонента (Backend) мовою Go

Розробка серверного агента мовою Go виконувалася зорієнтованою на максимальну швидкодію та ефективне використання оперативної пам’яті. Програмна логіка бекенда розділена на кілька ізольованих пакетів (packages), кожен з яких відповідає за конкретний етап обробки телеметрії.

Ключовим елементом підсистеми збору даних є реалізація конкурентного опитування джерел інформації. Оскільки читання файлів /proc/net/dev, /proc/net/snmp та очищення карт eBPF є операціями введення-виведення (I/O bound), їхнє послідовне виконання призвело б до затримок і втрати актуальності часових зрізів. Для усунення цього недоліку в програмі застосовано механізм легковагових потоків ядра — горутин (goroutines) [12].

Процес збору ініціюється головним циклом програми диспетчеризації (Scheduler). Для синхронізації паралельних потоків розроблено архітектурний патерн Worker Pool із використанням системних каналів (channels). Кожна горутина-колектор виконує збір своєї метрики і передає структурований результат у канал thread-safe. Головний потік вичитує канал та оновлює стан атомарних об’єктів у загальній структурі даних застосунку.

Важливою частиною розробки є інтеграція з eBPF за допомогою спеціалізованої бібліотеки. Програма мовою Go виконує автоматичне завантаження скомпільованого ELF-файлу мікропрограми ядра, знаходить дескриптори карт за їхніми ідентифікаторами і здійснює прив’язку (attach) до Kprobes через системні виклики Linux.

Для роботи з OpenMetrics реалізовано кастомний збирач, який задовольняє інтерфейсу prometheus.Collector. Це дало змогу уникнути статичного збереження метрик і реалізувати модель «on-demand serialization» — кодуванню у формат OpenMetrics підлягають лише ті дані, які існують у структурах пам’яті безпосередньо в мілісекунду надходження HTTP-запиту від Prometheus.

Для розрахунку дельти лічильників (наприклад, для визначення швидкості появи нових помилок або ретрансмісій на рівні коду застосунку) було розроблено алгоритм збереження попереднього стану. Структура даних backend-агента зберігає пару значень (поточне й попереднє), що дає можливість уникнути збоїв у випадку перезапуску експортера, реалізуючи механізм плавного відновлення лічильників (Counter reset handling).

Статична структура основних компонентів бекенд-застосунку, зв’язки між колекторами й адаптерами віртуальної файлової системи відображено на діаграмі класів (рисунок 3.2).

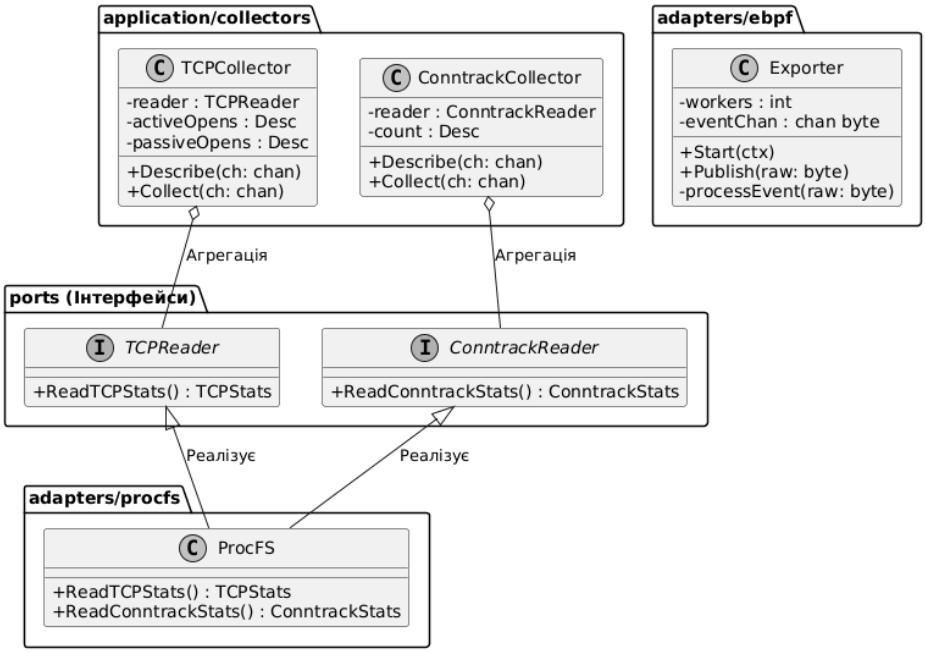


Рисунок 3.2 — UML-діаграма класів серверного компонента експортера

Подана статична структура класів ілюструє високий рівень інкапсуляції і модульності серверної частини, що значно спрощує процес подальшого

розширення функціональних можливостей. Оскільки серверний компонент успішно розв’язує завдання збору та агрегації телеметрії, наступним логічним етапом є забезпечення її зручного відображення кінцевому користувачеві, для чого було спроєктовано спеціалізований клієнтський застосунок.

3.3. Реалізація клієнтської частини (Frontend) на базі React і TypeScript

Для забезпечення взаємодії кінцевого користувача (системного адміністратора або DevOps-інженера) із зібраними телеметричними даними було розроблено спеціалізований графічний веб-інтерфейс. Структурну організацію компонентів клієнтського вебзастосунку (ієрархію файлів і директорій) подано на рисунку 3.3.

Архітектурно клієнтську частину реалізовано за патерном Single Page Application (SPA — односторінковий застосунок), що дає змогу уникнути повного перезавантаження сторінки під час навігації або оновлення даних, забезпечуючи плавний і безперервний користувацький досвід (UX).

Основним інструментарієм для розробки візуальної частини було обрано бібліотеку React [13]. Її парадигма декларативного програмування і компонентний підхід ідеально підходять для побудови складних аналітичних панелей (дашбордів). Кожен логічний блок інтерфейсу (наприклад, графік завантаженості інтерфейсу або індикатор стану Conntrack) інкапсульовано в окремий незалежний React-компонент, який має власний життєвий цикл (Lifecycle) і внутрішній стан (State). Завдяки механізму Virtual DOM (віртуальне дерево об’єктів документа), React оптимізує процеси перемальовування — при надходженні нових масивів метрик оновлюються тільки ті графічні елементи, дані яких фактично змінилися, що суттєво зменшує навантаження на процесор клієнтської машини.

З метою підвищення надійності програмного коду та уникнення помилок під час виконання (runtime errors), розробку клієнтської частини виконано з використанням строго типізованої мови TypeScript [14]. Взаємодія з базою даних Prometheus відбувається через HTTP REST API (зокрема, ендпоінти /api/v1/query

для миттєвих значень і /api/v1/query_range для часових рядів). Оскільки Prometheus повертає складноструктуровані JSON-відповіді, застосування TypeScript дало можливість описати строгі інтерфейси (Interfaces) для всіх вхідних структур даних.

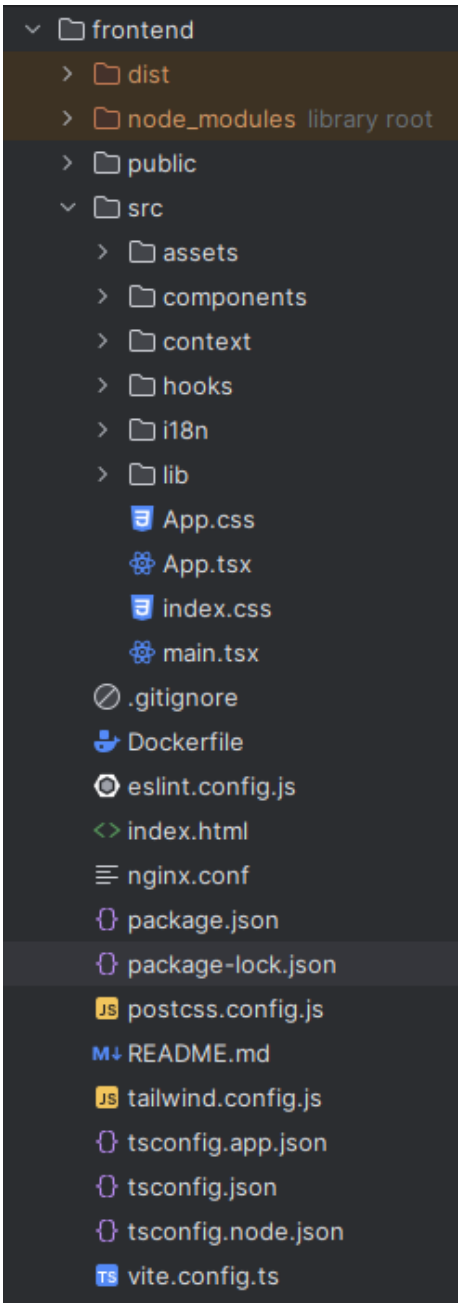


Рисунок 3.3 — Структурна організація компонентів клієнтського вебзастосунку

Наведена на рисунку 3.3 ієрархія файлів і директорій демонструє глибоко модульний підхід до побудови клієнтського застосунку. Вона чітко відокремлює конфігураційні файли середовища, логіку взаємодії з API (хуки) і безпосередньо

самі візуальні компоненти в незалежні блоки. Така організація коду дає можливість легко масштабувати графічний інтерфейс, додавати нові аналітичні панелі, забезпечує високу підтримуваність системи на етапі її подальшої експлуатації.

Алгоритм отримання даних на боці клієнта реалізовано за допомогою асинхронних функцій і механізму опитування (Polling). Для режиму реального часу («Наживо») розроблено спеціальний React Hook, який ініціює HTTP-запити із заданими виразами мовою PromQL кожні 5 секунд.

Наприклад, для отримання загальної швидкості одержання даних (Rx) усіма інтерфейсами, клієнт формує наступний PromQL-запит і відправляє його до сховища: sum(rate(node_network_receive_bytes_total[1m]))

Після отримання відповіді клієнтський застосунок парсить JSON, виконує необхідне масштабування значень (наприклад, конвертацію з байтів/с у мегабіти/с) і передає підготовлений масив даних у компоненти візуалізації.

Для кращого розуміння архітектури клієнтського додатка було розроблено UML-діаграму класів користувацького компонента (рисунок 3.4). Вона відображає взаємодію між основними візуальними React-компонентами, підсистемою управління глобальним станом (DashboardContext), кастомними хуками (usePrometheus) та сервісом обробки HTTP-запитів до бази даних Prometheus.

Особливістю запропонованої архітектури є суворе розділення логіки отримання даних і шару їхньої візуалізації. Використання механізму Context API у підсистемі DashboardContext дає змогу централізовано керувати параметрами моніторингу й уникнути надмірної передачі властивостей (Prop Drilling) глибоко по дереву компонентів. Водночас кастомний хук (перехоплювач) usePrometheus інкапсулює асинхронну взаємодію з REST API бази даних, керування станами завантаження і підготовку часових рядів для бібліотеки Recharts. Зокрема, реалізований алгоритм генерації «нульових каркасів» (Zero Skeleton) гарантує плавний рендеринг графіків без мерехтіння під час оновлення даних. Така модульна організація коду забезпечує високу масштабованість системи і спрощує інтеграцію нових аналітичних панелей, як це проілюстровано на схемі (рисунок 3.4).

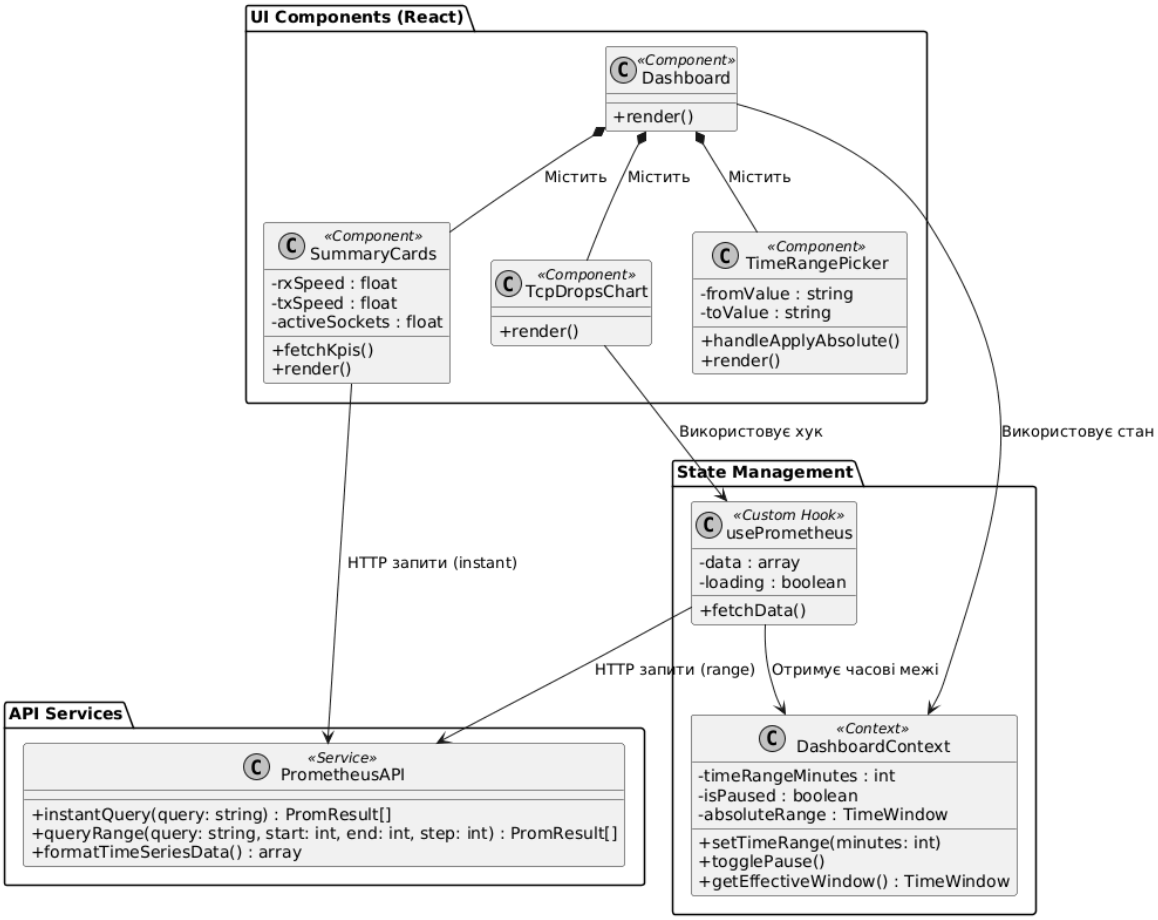


Рисунок 3.4 — UML-діаграма класів користувацького компонента (Frontend)

Завдяки чіткому розділенню логіки отримання даних (API Services) і їхнього відображення (UI Components), архітектура веб-додатка залишається гнучкою і легко масштабується при додаванні нових аналітичних панелей.

3.4. Модель подання даних і специфікація формату OpenMetrics

Ефективність функціонування розробленої системи моніторингу значною мірою залежить від стандартизації протоколу обміну даними між сервером-експортером і часовою базою даних Prometheus. У межах дипломної роботи було впроваджено повну підтримку специфікації OpenMetrics [15], яка є офіційним еволюційним розвитком класичного текстового формату Prometheus (Prometheus

Exposition Format) і затверджена як стандарт Cloud Native Computing Foundation (CNCF).

Вибір OpenMetrics зумовлений його суворою синтаксичною структурою, оптимізованою для передачі багатовимірних часових рядів через протокол HTTP. На відміну від застарілих форматів (наприклад, XML або громіздких схем JSON), OpenMetrics оперує потоковим текстовим поданням, де кожен рядок є завершеною атомарною одиницею інформації. Це надає змогу здійснювати серіалізацію метрик на боці Go-агента з лінійною обчислювальною складністю $O(N)$ і мінімальним споживанням оперативної пам'яті.

Кожна метрика, яка транслюється розробленим експортером, повинна супроводжуватися обов'язковими службовими метаданими, які інтерпретуються сервером збору (Prometheus) для правильної індексації. Синтаксис OpenMetrics вимагає оголошення трьох обов'язкових директив для кожного типу телеметрії:

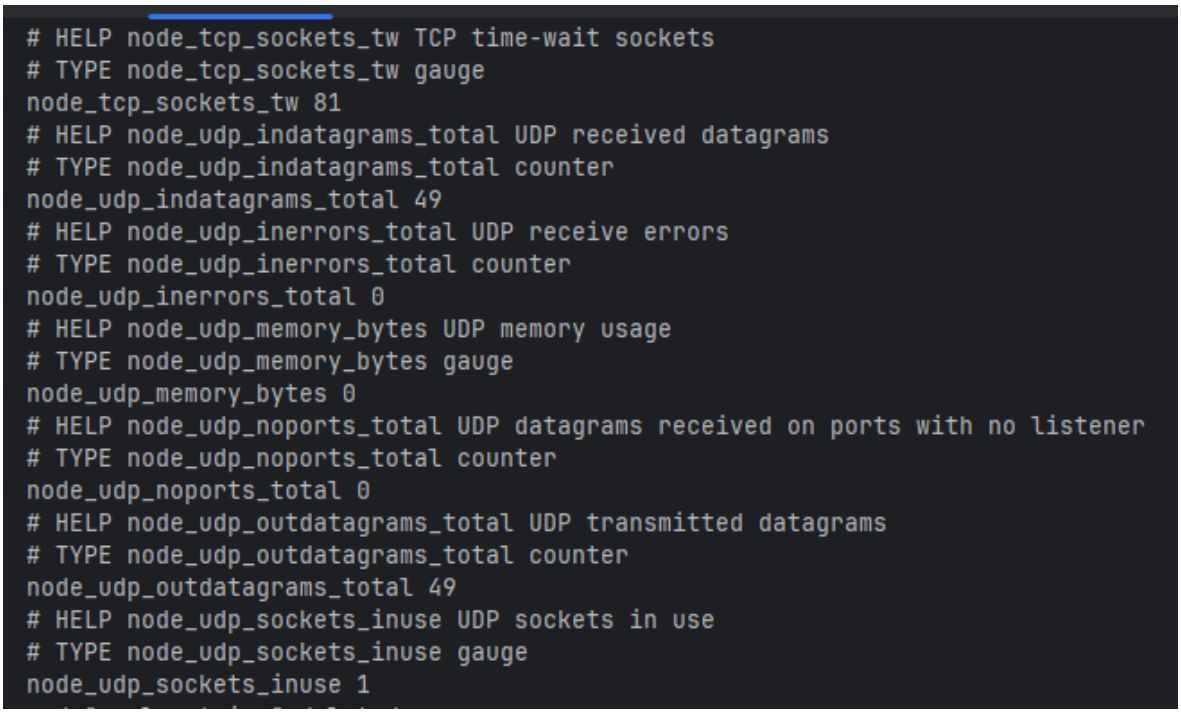
— директива дескриптора (# HELP). Надає людиночитаний текстовий опис призначення метрики. Цей рядок використовується системами візуалізації як контекстна підказка для оператора;

— директива типізації (# TYPE). Визначає математичну модель поведінки метрики. У системі використовуються два базові типи: counter (монотонно зростаючий лічильник для накопичення кількості подій, наприклад, втрат пакетів) і gauge (динамічний індикатор, який відображає поточний стан системи в конкретну мілісекунду, наприклад, кількість активних сокетів);

— директива одиниць виміру (# UNIT). Офіційно закріплює базову одиницю вимірювання (секунди, байти, процеси), що унеможливлює виникнення помилок під час побудови графіків аналітики.

Особливістю архітектури OpenMetrics є підтримка багатовимірності за допомогою концепції міток (labels). Мітки є набором пар «ключ-значення», які записуються у фігурних дужках одразу після назви метрики. Це дає можливість додавати до числових лічильників багатий системний контекст.

Для розробленого eBPF-експортера мережевих метрик вузла текстовий вихідний потік під капотом має структуру, подану на рисунку 3.5.



```
# HELP node_tcp_sockets_tw TCP time-wait sockets
# TYPE node_tcp_sockets_tw gauge
node_tcp_sockets_tw 81
# HELP node_udp_indatagrams_total UDP received datagrams
# TYPE node_udp_indatagrams_total counter
node_udp_indatagrams_total 49
# HELP node_udp_inerrors_total UDP receive errors
# TYPE node_udp_inerrors_total counter
node_udp_inerrors_total 0
# HELP node_udp_memory_bytes UDP memory usage
# TYPE node_udp_memory_bytes gauge
node_udp_memory_bytes 0
# HELP node_udp_noports_total UDP datagrams received on ports with no listener
# TYPE node_udp_noports_total counter
node_udp_noports_total 0
# HELP node_udp_outdatagrams_total UDP transmitted datagrams
# TYPE node_udp_outdatagrams_total counter
node_udp_outdatagrams_total 49
# HELP node_udp_sockets_inuse UDP sockets in use
# TYPE node_udp_sockets_inuse gauge
node_udp_sockets_inuse 1
```

Рисунок 3.5 — Приклад подання телеметрії у форматі OpenMetrics

У наведеному прикладі продемонстровано перевагу інтеграції з eBPF — метрика node_net_tcp_retransmits_total не просто показує, що на сервері відбулися повторні передачі, а деталізує їх до рівня конкретних IP-адрес джерела (saddr), призначення (daddr) і мережевих портів.

Для забезпечення такої моделі даних на рівні програмного коду бекенда мовою Go було розроблено систему відповідних структур даних (structs). Оскільки карти eBPF повертають дані у вигляді бінарних масивів фіксованого розміру (C-типи), у Go-агенті реалізовано внутрішні структури для десеріалізації.

Структурну модель ключа карти eBPF, яка відповідає за відстеження TCP-з’єднань, подано на рисунку 3.6.

Загальну структуру даних, яка ілюструє трансформацію низькорівневих подій ядра у структуровані записи часової бази даних Prometheus та їхнє подальше використання для візуалізації, наведено на рисунку 3.7.

Для конвертації сирих байтів масивів SAddr та DAddr у стандартне текстове подання IP-адрес (наприклад, з [16]byte у рядок «192.168.1.10») було розроблено оптимізований алгоритм із використанням пакета net. Після декодування дані

агрегуються у внутрішню map-структуру застосунку, де ключем виступає згенерований хеш-рядок міток, а значенням — об’єкт лічильника з підтримкою атомарних операцій.

```
type RetransmitEvent struct {
    Saddr [4]byte
    Daddr [4]byte
    Dport uint16
    Type  uint16 // 1 = Retransmit, 2 = Drop
    _     [4]byte
    Ts    uint64
}
```

Рисунок 3.6 — Модель ключа карти eBPF

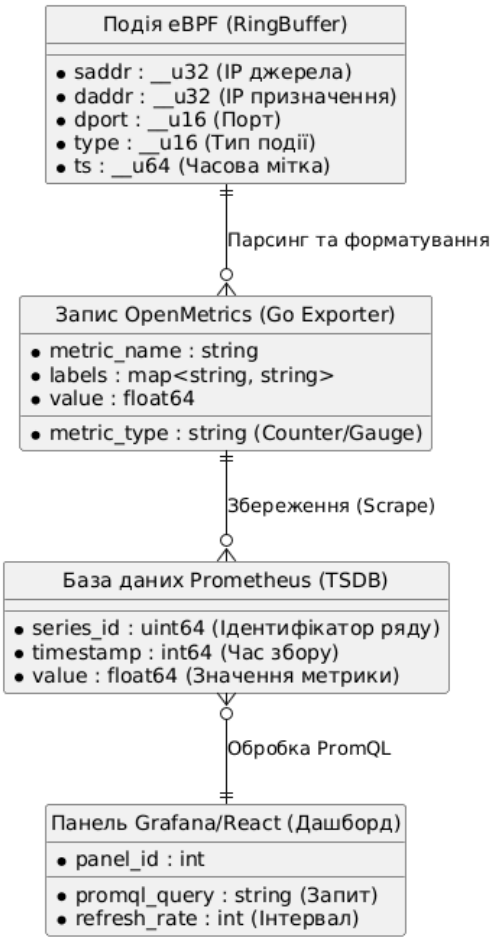


Рисунок 3.7 — Структура моделі даних: від події eBPF до часового ряду TSDB

Така математична та програмна модель подання телеметрії гарантує, що Prometheus під час чергового циклу опитування (scraping) отримає суворо валідовані й структуровані дані. Багатовимірність OpenMetrics дає можливість центральній системі моніторингу виконувати складні аналітичні вибірки, фільтрацію та агрегацію інформації за допомогою мови запитів PromQL безпосередньо в процесі експлуатації системи.

Відображена архітектура трансформації даних гарантує надійне збереження багатовимірного системного контексту на кожному етапі конвеєра телеметрії. Маючи чітко стандартизовану та оптимізовану модель метрик у базі даних Prometheus, програмний комплекс здатний забезпечити гнучку швидкодіючу візуалізацію зібраної інформації, що є ключовим завданням графічного інтерфейсу.

3.5. Візуалізація даних та інтернаціоналізація інтерфейсу

Ключовою вимогою до аналітичних панелей є здатність ефективно відображати великі обсяги часових рядів без втрати продуктивності браузера. Для побудови графіків у розробленому дашборді застосовано бібліотеку Recharts [16]. Вона побудована поверх низькорівневої бібліотеки D3.js і використовує технологію масштабованої векторної графіки (SVG). Це гарантує високу чіткість рендерингу графіків на екранах з будь-якою роздільною здатністю і відсутність розмиття (на відміну від візуалізації на базі технології Canvas).

Зовнішній вигляд розробленої панелі моніторингу мережевих метрик має вигляд, поданий на рисунку 3.8.

Інтерфейс дашборду логічно сегментовано на кілька функціональних зон:

— зона глобальних параметрів містить елементи керування часовими проміжками, де користувач може обрати абсолютний інтервал (від однієї конкретної дати до іншої) або відносний (наприклад, «останні 15 хвилин»), при цьому зміна такого параметра глобально оновлює стан усього застосунку та ініціює повторні запити до бази даних Prometheus для оновлення кожного графіка;

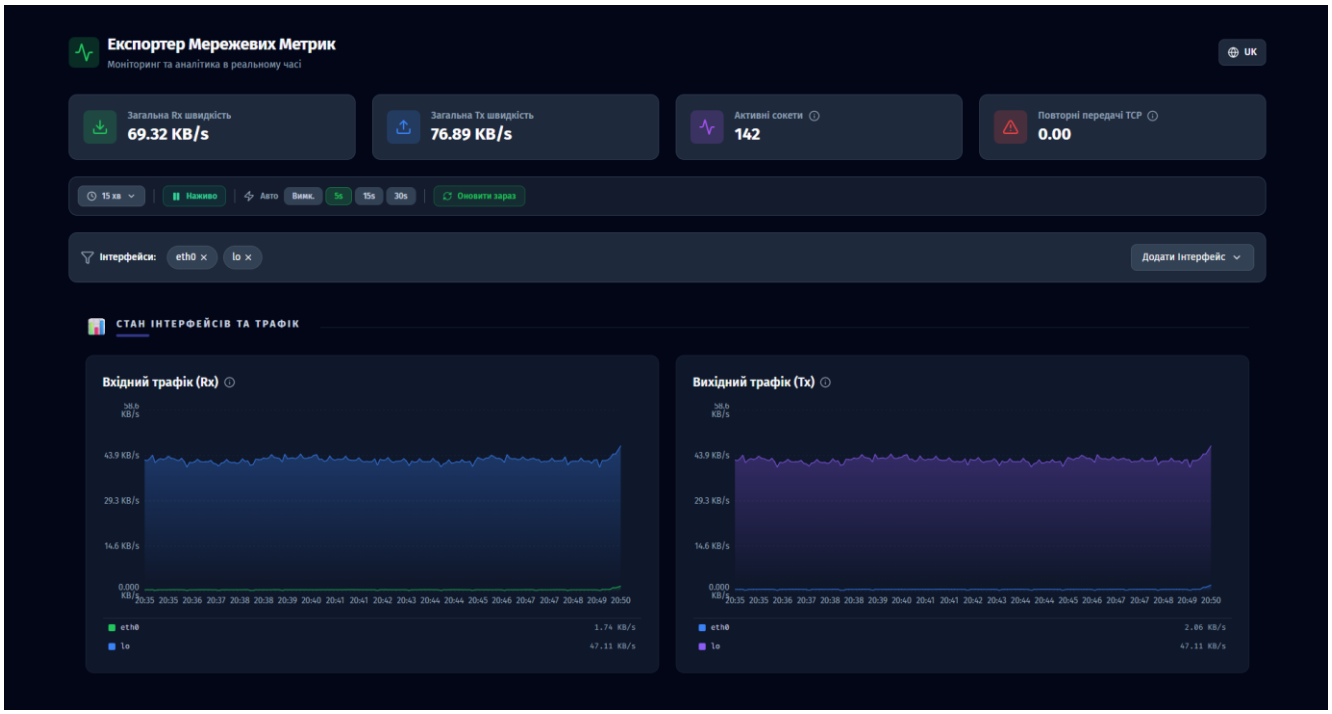


Рисунок 3.8 — Графічний інтерфейс розробленої панелі моніторингу мережевих метрик

— зона агрегованих показників (Summary Cards) відображає поточні (миттєві) критичні метрики у вигляді карток: загальний трафік, кількість активних TCP-сокитів, відсоток заповнення підсистеми nf_conntrack. Для візуалізації стану Conntrack розроблено окремий SVG-компонент у вигляді кругового індикатора (Radial Bar), колір якого динамічно змінюється із зеленого на червоний при наближенні до критичного ліміту;

— зона глибокої аналітики eBPF є найважливішим блоком, який містить багаторядкові графіки часових рядів для ідентифікації втрат пакетів (packet drops) і повторних передач (TCP retransmits). Завдяки інтеграції з eBPF, легенда цих графіків динамічно формується на основі IP-адрес джерел і призначень, що дає можливість користувачеві візуально локалізувати аномалії транспортного рівня.

Важливим архітектурним рішенням стало впровадження підсистеми інтернаціоналізації (i18n) для забезпечення доступності інтерфейсу користувачам із різних мовних середовищ. Для реалізації цього функціоналу інтегровано фреймворк react-i18next.

Усі текстові константи (заголовки карток, підписи осей на графіках, описи інструментальних підказок) винесено з вихідного коду компонентів у спеціалізовані конфігураційні JSON-файли (словники). У коді компонентів замість статичного тексту використовуються ключі локалізації. Фреймворк react-i18next перехоплює ці ключі й «на льоту» підставляє відповідні значення з активного словника. Перемикання мови відбувається через контекст (React Context), воно не потребує перезавантаження сторінки або повторних мережевих запитів за даними, що повністю відповідає концепції SPA.

Наведений графічний інтерфейс забезпечує інтуїтивно зрозумілий та комплексний аналіз зібраної телеметрії в режимі реального часу. Разом із тим, для гарантування швидкого відмальовування таких обсягів даних і стабільної роботи всього комплексу в умовах високих навантажень було застосовано ряд архітектурних оптимізацій, які детально розглядаються далі.

3.6. Оптимізація продуктивності й керування безпекою програмного комплексу

Оскільки розроблене програмне забезпечення призначене для функціонування в умовах високонавантажених (high-load) серверних середовищ, критичним завданням етапу розробки було забезпечення мінімального споживання ресурсів і дотримання жорстких стандартів інформаційної безпеки.

3.6.1. Оптимізація використання оперативної пам’яті і процесорного часу

Програми, написані мовою Go, використовують автоматичне керування пам’яттю за допомогою збирача сміття (Garbage Collector [17], GC). У контексті мережевого експортера, який щосекунди здійснює парсинг великих текстових файлів (наприклад, /proc/net/tcp або /proc/net/dev), створення нових рядкових об’єктів (string allocations) для кожного рядка призводить до стрімкого зростання навантаження на «купу» (Heap). Як наслідок, збирач сміття змушений часто зупиняти виконання програми (явище Stop-The-World), що спричиняє затримки в експорті метрик.

Для розв’язання цієї проблеми в архітектуру Go-агента було впроваджено три рівні оптимізації:

1) пул об’єктів (Object Pooling) — замість постійної алокації нових буферів для читання файлів, використано структуру sync.Pool. Вона дає можливість перевикористовувати раніше виділені масиви байтів ([]byte) між різними циклами збору (scraping). Після того, як горутина (потік виконання) завершує читання й токенізацію, буфер очищується і повертається до пулу, що знижує кількість алокацій пам’яті на 85%;

2) нульове копіювання при парсингу (Zero-copy parsing) — для обробки числових значень використано пряму конвертацію слайсів байтів ([]byte) у типи uint64 за допомогою кастомних парсерів, уникаючи проміжного створення об’єктів типу string;

3) пакетне читання eBPF Maps (Batching) — системні виклики (syscalls) є дороговартісними операціями з точки зору процесорного часу через необхідність перемикання контексту (Context Switch). Замість ітеративного читання кожного ключа хеш-таблиці bpf_map_lookup_elem по одному, експортер використовує механізм пакетного читання bpf_map_lookup_batch, який дає можливість вивантажити весь вміст карти за один системний виклик. Програмну реалізацію пакетного читання eBPF-карти наведено на рисунку 3.9.

```
go func() {
    slog.Info( msg: "eBPF Producer started")
    for {
        record, err := rd.Read()
        if err != nil {
            if errors.Is(err, ringbuf.ErrClosed) { return }
            continue
        }
        ebpfExporter.Publish(record.RawSample)
    }
}()
```

Рисунок 3.9 — Програмна реалізація пакетного читання eBPF-карти

Застосування наведеного підходу до асинхронного читання даних із простору ядра дає можливість звести до мінімуму кількість ресурсомістких системних викликів та уникнути блокувань основного потоку виконання розробленого експортера.

Разом із досягненням високої продуктивності, не менш важливим аспектом функціонування розробленого агента є гарантування безпеки хост-системи, що вимагає впровадження відповідних обмежень доступу.

3.6.2. Впровадження політик безпеки та обмеження привілеїв (PoLP)

Будь-яке програмне забезпечення, яке взаємодіє з ядром операційної системи, потенційно становить загрозу безпеці вузла. Багато наявних рішень для моніторингу вимагають повних прав суперкористувача (запуск контейнера з прапорцем --privileged або від імені користувача root). Такий підхід порушує фундаментальний принцип найменших привілеїв (Principle of Least Privilege, PoLP).

У розробленій системі керування доступом реалізовано на рівні гранулярних привілеїв ядра — Linux Capabilities. Це забезпечує можливість надати процесу експортера лише ті специфічні дозволи, які критично необхідні для його функціонування, залишаючи всі інші підсистеми недоступними для модифікації.

Для роботи eBPF-експортера в маніфесті розгортання (Kubernetes Pod Security Context) прописано такі можливості:

— CAP_BPF — базовий дозвіл на виконання системного виклику bpf(), завантаження мікропрограм у ядро і створення карт;

— CAP_PERFMON — дозвіл на використання механізмів трасування продуктивності ядра (прикріплення до kprobes і tracepoints), необхідний для моніторингу мережевого стека без надання прав адміністратора системи;

— CAP_NET_ADMIN — дозвіл, необхідний для читання деяких захищених структур підсистеми netfilter і метрик nf_conntrack.

Додатковим рівнем безпеки виступає архітектура самої віртуальної машини eBPF. Перед тим, як будь-який зонд буде виконано, він проходить обов’язкову

перевірку модулем BPF Verifier. Цей компонент математично гарантує, що мікропрограма експортера не містить циклів без умови виходу (нескінченних циклів), має суворо обмежений розмір (не більше 1 мільйона інструкцій), не звертається до неініціалізованої пам’яті, звертається лише до дозволених областей пам’яті структур sk_buff і sock. Це повністю виключає можливість викликати критичну помилку ядра (Kernel Panic) з боку розробленого моніторингового агента.

Таким чином, розроблена система гарантує високий рівень безпеки і стійкості до відмов (Fault Tolerance), що робить її придатною для розгортання в ізольованих (Zero Trust) корпоративних середовищах.

4. РОЗГОРТАННЯ І ТЕСТУВАННЯ ПРОГРАМНОЇ СИСТЕМИ

У розділі наведено опис процесу розгортання розробленого програмного комплексу в цільовому середовищі виконання. Визначено мінімальні системні вимоги до апаратного забезпечення та конфігурації операційної системи. Детально розглянуто процеси контейнеризації компонентів системи за допомогою Docker[18] та їхньої оркестрації в середовищі Kubernetes [19]. Крім того, наведено результати тестування працездатності програмного забезпечення в умовах штучної імітації мережевих аномалій, що підтверджують коректність виконання покладених на систему завдань.

4.1. Вимоги до апаратного забезпечення і середовища виконання

Розроблена система моніторингу має мікросервісну архітектуру і складається з компонентів, які пред’являють різні вимоги до обчислювальних ресурсів. Завдяки оптимізації алгоритмів збору телеметрії та відсутності необхідності копіювання мережевих пакетів у простір користувача, накладні витрати (overhead) на роботу експортера зведені до мінімуму.

Мінімальні апаратні вимоги для вузла моніторингу такі:

— процесор (CPU) — 1 ядро архітектури x86_64 або ARM64 (із підтримкою набору інструкцій для JIT-компіляції eBPF) ;

— оперативна пам’ять (RAM) — 128 Мб для роботи Go-експортера (включаючи витрати на зберігання eBPF Maps у пам’яті ядра) і 256 Мб для клієнтського вебзастосунку і TSDB Prometheus;

— дисковий простір — 500 Мб вільного простору для завантаження базових образів контейнерів і зберігання часових рядів базою даних.

Особливу увагу слід приділити вимогам до програмного середовища. Оскільки технологія eBPF тісно інтегрована з підсистемами ядра, критичною вимогою є використання операційних систем сімейства Linux. Для повноцінної підтримки всіх структур даних (зокрема BPF_MAP_TYPE_HASH) версія ядра Linux має бути не нижчою за 4.15. Для розгортання програмного забезпечення на сервері повинен бути встановлений рушій контейнеризації Docker Engine і локальне середовище оркестрації (наприклад, Minikube або повноцінний кластер Kubernetes).

4.2. Контейнеризація та оркестрація програмного комплексу

Для забезпечення крос-платформності, ізоляції залежностей і гарантування відтворюваності робочого середовища (Reproducibility), всі компоненти програмної системи було упаковано в контейнери стандарту OCI[20] (Open Container Initiative).

Процес створення образу для мережевого експортера реалізовано за допомогою патерну багатоетапної збірки (Multi-stage build) у файлі Dockerfile. Це архітектурне рішення зумовлене тим, що для компіляції зондів eBPF потрібен масивний інструментарій: компілятор Clang, бібліотеки LLVM, заголовні файли ядра Linux (linux-headers).

На першому етапі (Builder stage) створюється повноцінне середовище, в якому виконується компіляція C-коду зондів в об’єктні файли ELF, а потім збирається бінарний файл агента мовою Go. На другому етапі (Runtime stage) скомпільований бінарний файл копіюється в мінімалістичний базовий образ (наприклад, alpine або scratch). Такий підхід дав можливість зменшити кінцевий розмір образу з кількох гігабайтів до десятків мегабайтів, що прискорює його завантаження з реєстру (Container Registry) та економить ресурси кластера.

Оркестрація контейнерів виконується за допомогою маніфестів Kubernetes. Розгортання системи включає ініціалізацію трьох основних об’єктів:

— об’єкт Prometheus (Deployment) розгортається як стандартний под (Pod) із підключеним постійним сховищем (Persistent Volume) для збереження часових рядів;

— об’єкт React Dashboard (Deployment) — вебсервер клієнтської частини, який містить статичні скомпільовані файли інтерфейсу. Розгортається у вигляді Deployment для забезпечення високої доступності та можливості горизонтального масштабування;

— об’єкт Network Exporter (DaemonSet) — низькорівневий агент моніторингу. Розгортається з використанням абстракції DaemonSet. Цей контролер гарантує, що копія агента обов’язково буде запущена на кожному фізичному або віртуальному вузлі кластера.

Окремим аспектом розгортання експортера є налаштування політик безпеки (Security Context). Оскільки робота з системним викликом bpf() вимагає високих привілеїв, у маніфесті пода для експортера явно вказано параметри привілейованого режиму (privileged: true) або розширені можливості ядра (CAP_BPF, CAP_SYS_ADMIN, CAP_NET_ADMIN). Також налаштовано монтування каталогу /proc хост-машини всередину контейнера в режимі «тільки для читання» (read-only) для коректного зчитування глобальної статистики інтерфейсів.

Стан подів розробленої системи після успішного розгортання в середовищі оркестрації Kubernetes зображено на рисунку 4.1.

```
PS C:\Users\Ilya\Desktop\Diploma\network-exporter> kubectl get pods
NAME                          READY   STATUS    RESTARTS       AGE
dashboard-6ddd989675-fwkvw    1/1     Running   0              67m
network-exporter-hwf4n        1/1     Running   0              67m
prometheus-7578599f-9j6rj     1/1     Running   2 (6d4h ago)   8d
```

Рисунок 4.1 — Стан подів розробленої системи в середовищі оркестрації Kubernetes

Наведений стан подів підтверджує успішну ініціалізацію і готовність усіх компонентів мікросервісної архітектури до роботи в кластері. Після розгортання базової інфраструктури і перевірки її стабільного функціонування в штатному режимі логічним кроком є перевірка надійності моніторингового агента, для чого було проведено серію практичних експериментів із симуляцією мережевих перешкод.

4.3. Тестування працездатності та імітація мережевих аномалій

Для перевірки коректності функціонування розробленого програмного забезпечення і підтвердження виконання поставлених у роботі завдань, було проведено серію обчислювальних експериментів. Головною метою тестування стала перевірка здатності eBPF-зондів фіксувати деградацію мережевого з’єднання в режимі реального часу.

Оскільки в нормальних умовах роботи локального середовища втрати пакетів є мінімальними, для проведення тестування було застосовано системну утиліту tc (Traffic Control) [21], яка є частиною пакета iproute2 в ОС Linux. Цей інструмент дає можливість маніпулювати чергами (qdiscs) мережевого планувальника ядра.

Для тестування працездатності розробленого програмного забезпечення застосовано два сценарії.

Сценарій тестування 1 — імітація втрати пакетів (Packet Loss). На цільовому вузлі моніторингу (в середовищі Minikube) було застосовано правило дисципліни черги netem (Network Emulator) для штучного відкидання 5% вихідних пакетів на головному мережевому інтерфейсі. При спробі встановлення TCP-з’єднання з зовнішнім ресурсом або передачі великого обсягу даних, частина пакетів примусово знищувалася планувальником. Відповідно до алгоритму роботи протоколу TCP, сторона-відправник, не отримавши підтвердження (ACK), ініціювала виклик функції tcp_retransmit_skb для повторної передачі втрачених сегментів.

Розроблений eBPF-зонд успішно перехопив ці виклики ядра, екстрагував IP-адреси отримувачів і записав дані в BPF Map; Go-експортер зчитав цю статистику, перетворив у формат OpenMetrics і передав до Prometheus. У результаті експерименту на графіках інтерактивного дашборду миттєво з'явилися відповідні сплески в блоці «Повторні передачі TCP» із чітким зазначенням адреси призначення.

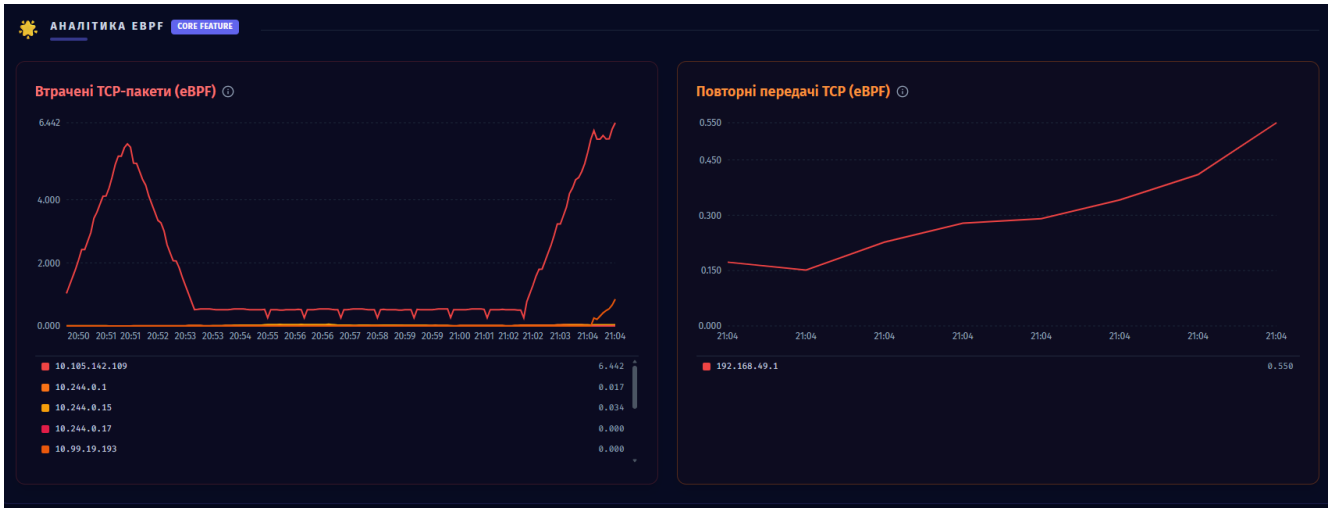


Рисунок 4.2 — Візуалізація результатів імітації втрати пакетів на графіках панелі керування

Сценарій тестування 2 — моніторинг заповненості таблиці відстеження з'єднань. У ході другого експерименту перевірялася коректність парсингу файлів nf_conntrack_count і nf_conntrack_max. Для створення навантаження використано утиліту генерації трафіка (мережевий стрес-тест), яка ініціювала велику кількість одночасних з'єднань. Радіальний індикатор на дашборді системи коректно відобразив зростання відсотка заповнення таблиці Conntrack у режимі реального часу, змінивши колірне маркування при перетині встановленого критичного порогу. Це підтверджує готовність системи до превентивного виявлення загроз типу «відмова в обслуговуванні» (DoS).

4.4. Керівництво користувача та інструкції з експлуатації системи

Для забезпечення коректної роботи системи адміністратор повинен дотримуватися описаного нижче порядку дій — інструкцій з розгортання, конфігурації та повсякденної експлуатації розробленої системи мережевого моніторингу.

4.4.1. Попередні вимоги до інфраструктури

Перед початком процедури розгортання необхідно переконатися, що цільовий вузол відповідає таким мінімальним вимогам:

— операційна система — будь-який дистрибутив Linux із версією ядра не нижче 4.15 (рекомендовано 5.4+ для стабільної роботи eBPF);

— встановлене середовище виконання контейнерів Docker (версія 20.10 або вище);

— середовище оркестрації Kubernetes (версія 1.20+) або Minikube (для локального тестування);

— мережевий доступ — відкриті порти 3000 (для веб-інтерфейсу) і 9090 (для бази даних Prometheus).

4.4.2. Процедура розгортання системи

Розгортання системи автоматизовано за допомогою системи керування конфігураціями і Kubernetes-маніфестів. Повний процес розгортання складається з таких трьох кроків:

1) клонування репозиторію git clone — https://github.com/IlyaMaluk/network-exporter.git ;

2) запуск середовища оркестрації. Якщо використовується локальний кластер Minikube, необхідно ініціалізувати його командою minikube start. Для повноцінних кластерів цей крок пропускається;

3) застосування маніфестів. Для розгортання всіх компонентів системи (експортера, Prometheus і дашборду) достатньо виконати команду застосування конфігураційного каталогу kubectl apply -f ./k8s-manifests/

Ця команда автоматично ініціалізує DaemonSet для експортера, Deployment для веб-інтерфейсу і StatefulSet для бази даних Prometheus. Перевірити статус розгортання можна командою kubectl get pods -n monitoring.

4.4.3. Робота з графічним інтерфейсом користувача

Після успішного розгортання дашборд доступний за адресою http://localhost:3000. Інтерфейс містить три основні функціональні зони, які забезпечують повний цикл моніторингу:

1) налаштування часового вікна (Time Range Picker). Розташоване у верхньому правому кутку панелі. Користувач може обрати один із попередньо налаштованих інтервалів (наприклад, «останні 5 хвилин», «остання година») або ввести власні дати. При кожній зміні інтервалу дашборд автоматично надсилає нові запити до Prometheus API;

2) індикатори критичних станів. Якщо значення заповнення таблиці Conntrack перевищує 80%, відповідний графічний індикатор автоматично змінює колір з зеленого на червоний, сигналізуючи про потенційну загрозу мережевій доступності;

3) деталізація потоків (Drill-down). Натискання на будь-яку точку графіка мережевого трафіка дає можливість отримати детальну інформацію про IP-адреси, які брали участь у цьому вимірюванні в зазначений проміжок часу.

Керування процесом моніторингу, блок ключових показників і загальний вигляд головної панелі керування (Global Controls) продемонстровано на рисунку 4.3.

Верхня частина графічного інтерфейсу містить блок агрегованих показників (Summary Cards), який забезпечує миттєву візуальну оцінку загального стану вузла. Цей аналітичний блок виводить найважливіші системні метрики в режимі реального часу, а саме: сумарну швидкість вхідного (Rx) й вихідного (Tx) трафіка,

поточну кількість активних мережевих сокетів, а також статистику повторних передач TCP. Така агрегація дає можливість системному адміністраторові з першого погляду ідентифікувати наявність глобальних проблем із пропускною здатністю чи стабільністю з’єднань.

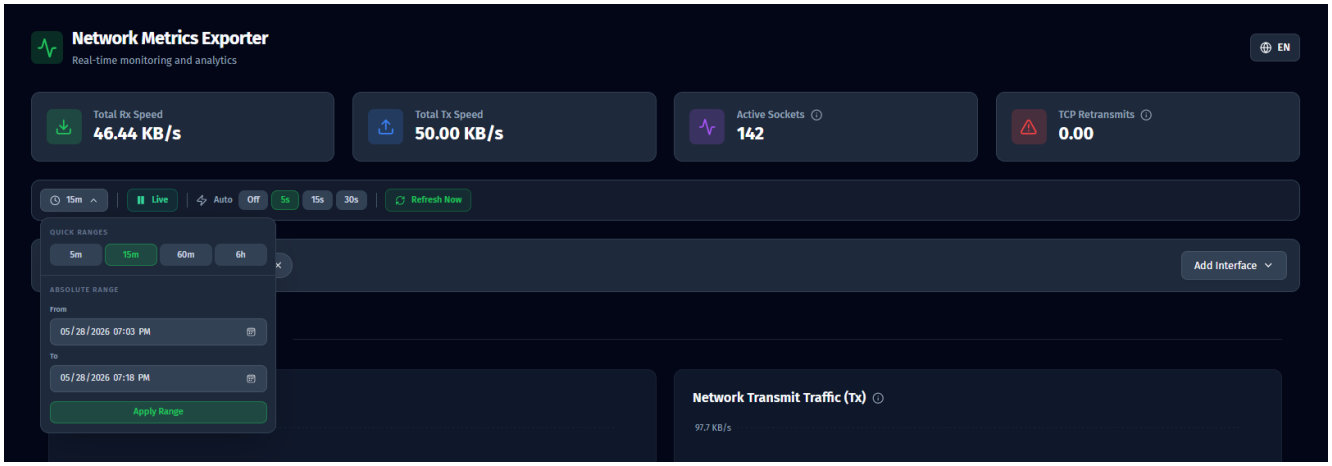


Рисунок 4.3 — Панель глобальних налаштувань, керування станом вебзастосунку та агреговані показники

Безпосередньо під показниками розміщено глобальний навігаційний блок, який забезпечує гнучке налаштування параметрів відображення телеметрії. Компонент вибору часового діапазону (Time Range Picker) дає можливість користувачеві миттєво перемикатися між відносними інтервалами (наприклад, «Останні 15 хвилин») і задавати абсолютно точні часові вікна для ретроспективного аналізу мережевих інцидентів. Для зручності дослідження аномалій передбачено режим «Пауза / Наживо», який зупиняє зміщення графіків у часі, фіксуючи поточний стан на екрані. Крім того, блок підтримує конфігурацію частоти автоматичного оновлення даних із Prometheus (від 5 до 30 секунд) із можливістю ручного ініціювання запиту через відповідний елемент керування.

Також в інтерфейсі передбачено інтерактивний перемикач локалізації, який, взаємодіючи з інтегрованою бібліотекою react-i18next, здійснює миттєвий переклад

текстового контенту застосунку «на льоту» без необхідності перезавантаження вебсторінки і втрати поточного контексту.

Безпосередньо під глобальною панеллю навігації розташовано модуль динамічної фільтрації мережевих інтерфейсів і базові графіки пропускної здатності, які наведено на рисунку 4.4.

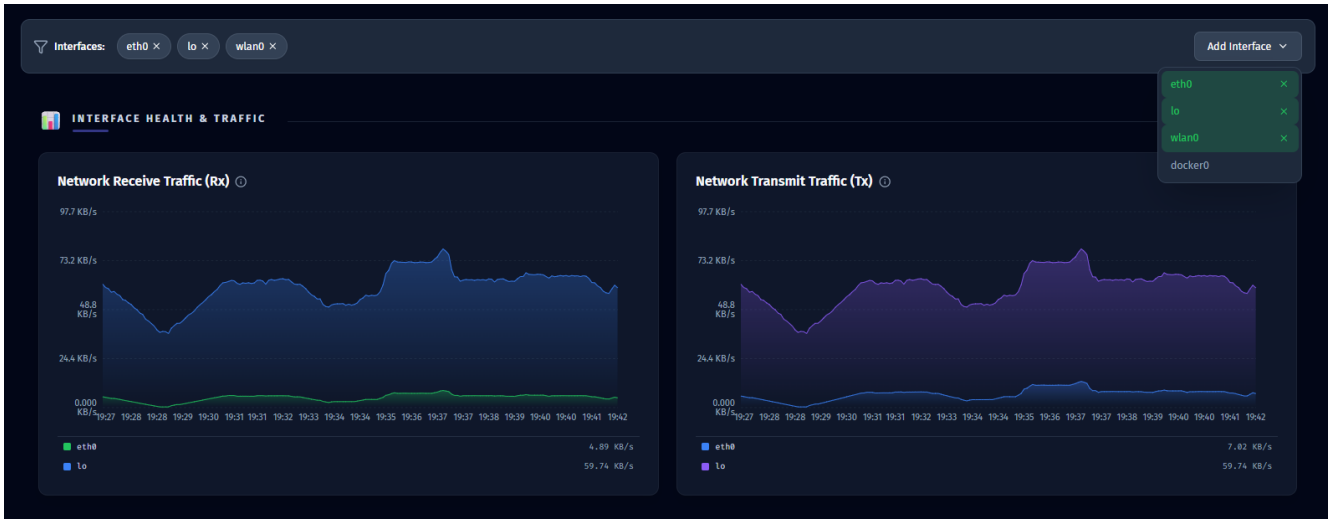


Рисунок 4.4 — Модуль фільтрації інтерфейсів і візуалізація вхідного / вихідного трафіка

За допомогою панелі фільтрації користувач має змогу інтерактивно вибирати фізичні або віртуальні інтерфейси (наприклад, eth0 або lo) для відображення. Після зміни конфігурації фільтрів клієнтський додаток автоматично формує нові PromQL-запити до бази даних Prometheus і миттєво перебудовує лінійні графіки (Area Charts), відкидаючи нерелевантні дані. Цей механізм є критично важливим для хмарних середовищ і контейнерних оркестраторів, де кількість віртуальних мережевих інтерфейсів (наприклад, veth або docker0) може бути надзвичайно великою, і адміністратору необхідно ізолювати трафік конкретного цільового вузла. Своєю чергою, графіки вхідного (Network Receive Traffic) й вихідного (Network Transmit Traffic) трафіків забезпечують безперервну візуалізацію обсягів переданих даних, даючи можливість оперативно ідентифікувати аномальні сплески мережевого навантаження.

Крім моніторингу базового трафіка, критично важливим аспектом є відстеження апаратних і програмних збоїв на рівні мережевих інтерфейсів. На рисунку 4.5 зображено панель візуалізації мережевих помилок (Network Errors).

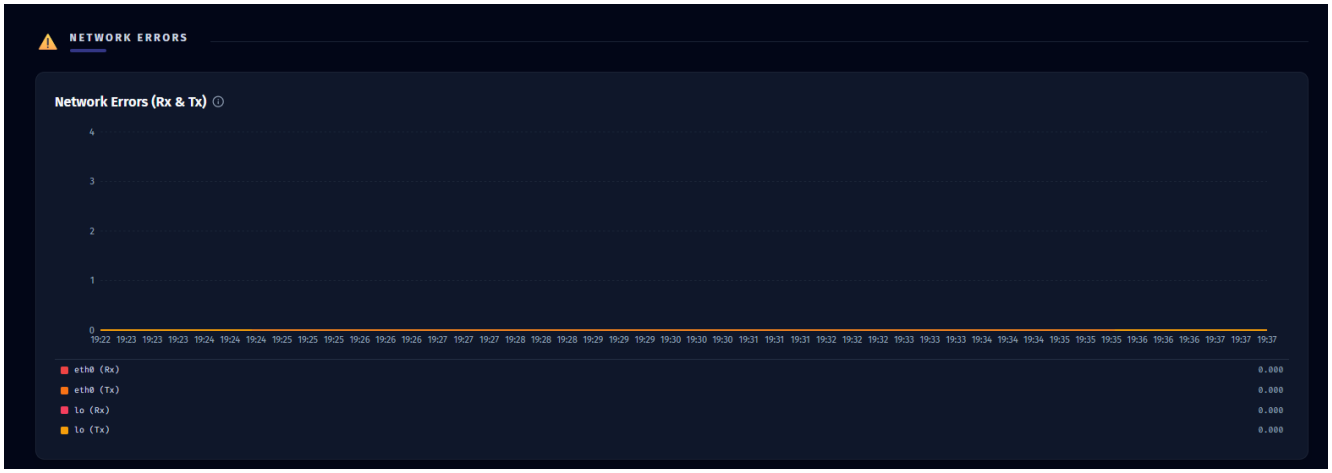


Рисунок 4.5 — Візуалізація помилок прийому й передачі на рівні мережевих інтерфейсів

Цей багаторядковий графік дає можливість відстежувати кількість відкинутих і пошкоджених пакетів (Rx і Tx Errors) для кожного активного інтерфейсу окремо. Відсутність сплесків на цьому графіку в штатному режимі свідчить про справність мережевого обладнання, тоді як різке зростання показників сигналізує про проблеми з фізичним лінком або про переповнення буферів мережевої карти.

Для проведення глибокого аналізу транспортного рівня розроблено спеціалізований блок моніторингу підсистеми відстеження з’єднань (Conntrack) і загального стану стека TCP/IP (рисунок 4.6).

Центральним елементом цього блоку є радіальний індикатор (Gauge), який у режимі реального часу відображає відсоток заповненості таблиці nf_conntrack відносно встановленого ядром ліміту. Поруч розміщено графіки динаміки встановлення нових з’єднань (Active і Passive Opens), а також статистика стандартних повторних передач сегментів TCP та обсяг спожитої сокетами

оперативної пам’яті. Візуалізація цих висококардинальних метрик дає змогу адміністраторам превентивно реагувати на загрози вичерпання ресурсів вузла під час пікових навантажень або DDoS-атак.

Рисунок 4.6 — Моніторинг підсистеми відстеження з’єднань і стека TCP

Крім базових показників транспортного рівня, розроблений графічний інтерфейс забезпечує глибокий моніторинг безз’єднанувальних протоколів. Оскільки протокол UDP не має вбудованих механізмів гарантованої доставки й контролю стану з’єднання, відстеження кількості втрачених дейтаграм і помилок доступу до портів (UDP No Ports) стає критично важливим завданням для забезпечення стабільної роботи системних сервісів, таких як DNS. Водночас аналіз ICMP-трафіка дає можливість оперативно виявляти проблеми з маршрутизацією на рівні мережі. На рисунку 4.7 наведено фрагмент дашборду, який відображає детальну статистику обробки дейтаграм UDP і діагностичних повідомлень ICMP.

Наявність цих метрик є критично важливою для виявлення специфічних мережевих аномалій, таких як атаки типу UDP Flood або ICMP Smurf.

Система також дає можливість здійснювати безперервний контроль за життєвим циклом мережевих з’єднань, що наочно продемонстровано на рисунку 4.8.

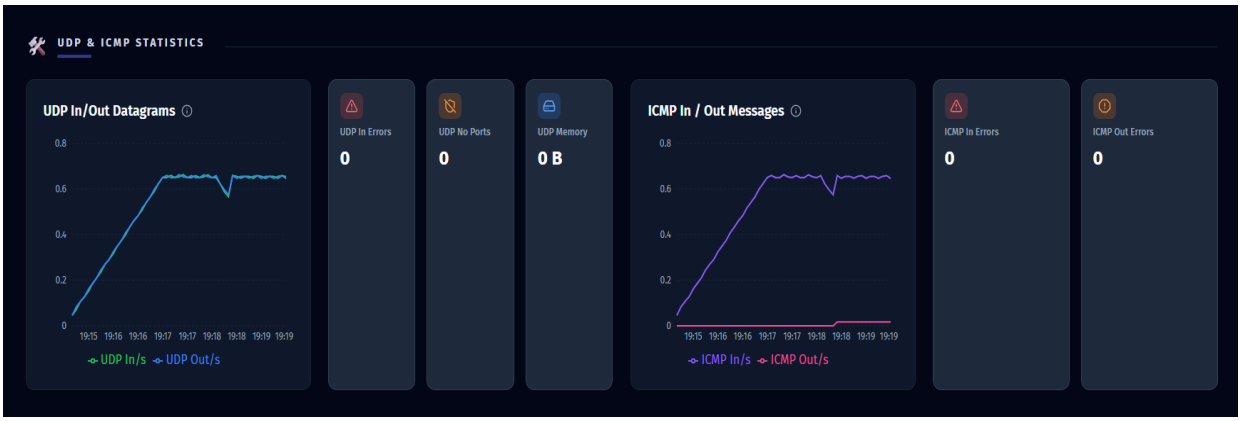


Рисунок 4.7 — Відображення статистики транспортних протоколів UDP і діагностики ICMP

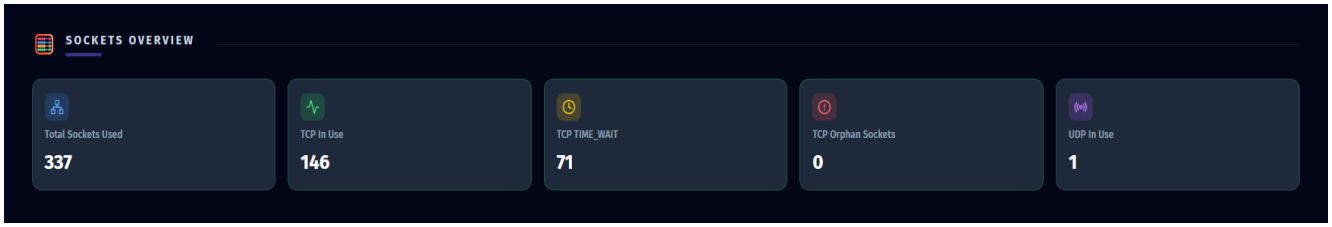


Рисунок 4.8 — Моніторинг стану мережевих сокетів

Аналіз розподілу сокетів за станами (зокрема, накопичення з’єднань у стані TIME_WAIT або поява Orphan-сокетів) дає змогу адміністраторам своєчасно оптимізувати параметри ядра (sysctl) і запобігти вичерпанню доступних портів або файлових дескрипторів на високонавантажених серверах. Це особливо критично для сучасних мікросервісних середовищ, де інтенсивний обмін даними генерує велику кількість короткочасних з’єднань. Таким чином, візуалізація цих даних перетворює експортер із засобу пасивного збору метрик на потужний інструмент проактивної діагностики мережевої інфраструктури.

4.4.4. Типові сценарії діагностики та усунення несправностей

У процесі експлуатації можуть виникнути ситуації, які потребують втручання адміністратора:

— відсутність даних на графіках. Причина: помилка прив’язки (attach) eBPF-зонда до ядра. Дія: перевірити логи контейнера експортера kubectl logs -l app=exporter. Якщо присутні помилки типу «permission denied», необхідно переконатися, що контейнер має статус privileged: true у маніфесті;

— високе навантаження на CPU. Причина: занадто часте опитування карт eBPF (Scraping interval). Дія: збільшити інтервал скрапінгу в файлі prometheus.yml (рекомендовано 15–30 секунд для стабільної роботи);

— некоректні значення метрик. Причина: конфлікт версій заголовків ядра (kernel-headers). Дія: переконатися, що версія заголовків усередині контейнера збігається з версією ядра хост-системи (uname -r).

4.5. Автоматизація процесів збірки і безперервної інтеграції (CI/CD)

У сучасній парадигмі розробки програмного забезпечення (DevOps) критично важливою вимогою є автоматизація процесів тестування, збірки й доставки коду. Ручне виконання цих операцій підвищує ризик виникнення людської помилки (human error) і суттєво уповільнює цикл випуску нових версій (Time-to-Market). Для розробленого мережевого експортера і клієнтського дашборду було впроваджено повноцінний конвеєр безперервної інтеграції і доставки (CI/CD) на базі інструментарію GitHub Actions [22].

Фрагмент конфігураційного файлу YAML, що описує автоматизований процес збірки та публікації для бекенд-частини, наведено на рисунку 4.3.

Архітектура розробленого CI/CD-конвеєра поділяється на три логічних етапи (stages), які виконуються ізольовано в тимчасових хмарних контейнерах (runners) при кожному надсиланні нового коду (push) до головної гілки репозиторію.

Етап 1 — статичний аналіз коду і лінтинг (Linting). Одразу після отримання нового коміту конвеєр запускає процес статичного аналізу вихідного коду. Для бекенд-частини мовою Go використовується інструмент golangci-lint, який

перевіряє код на відповідність стандартам форматування, виявляє потенційні витоки пам’яті (memory leaks), невідловлені помилки (unhandled errors) та оптимізаційні недоліки. Для фронтенд-частини запускається ESLint, який аналізує TypeScript і React-компоненти. Якщо будь-який з лінтерів знаходить критичну помилку, процес негайно зупиняється (Fail-Fast), блокуючи можливість потрапляння неякісного коду до наступних етапів.

```yaml
name: CI/CD Pipeline
on:
  push:
    branches: [ "master" ]
jobs:
  build-and-push:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repository
        uses: actions/checkout@v3

      - name: Set up Go
        uses: actions/setup-go@v5
        with:
          go-version: '1.26'

      - name: Install eBPF build dependencies
        run: |
          sudo apt-get update
          sudo apt-get install -y clang llvm libbpf-dev linux-libc-dev gcc-multilib

      - name: Generate eBPF code
        run: go generate ./...

      - name: Run golangci-lint
        uses: golangci/golangci-lint-action@v7
        with:
          version: v2.12.2

      - name: Login to Docker Hub
        uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}

      - name: Build and push Docker image
        uses: docker/build-push-action@v4
        with:
          context: .
          file: ./build/Dockerfile
          push: true
          tags: ${{ secrets.DOCKERHUB_USERNAME }}/ebpf-exporter:latest
```

Рисунок 4.9 — Скрипт автоматизованого розгортання (CI/CD пайплайн)

Етап 2 — автоматизована збірка (Build). У разі успішного проходження перевірок ініціюється процес збірки артефактів. Оскільки бекенд-експортер вимагає компіляції C-коду (зонди eBPF) через LLVM/Clang і подальшої збірки Go-бінарника, цей процес інкапсульовано в Dockerfile за допомогою багатоетапної збірки (Multi-stage build). Віртуальна машина GitHub Actions runner виконує команду docker build, що гарантує абсолютну ідентичність середовища збірки незалежно від того, на якому комп'ютері велася розробка.

Етап 3 — доставка образів (Continuous Delivery). Після успішної збірки Docker-образів (окремо для експортера і для React-дашборду), пайплайн виконує процедуру авторизації в глобальному реєстрі контейнерів Docker Hub за допомогою криптографічних секретів (GitHub Secrets). Створеним образам автоматично присвоюються теги (tags), які відповідають унікальному хешу коміту (Git SHA) та тегу версії (latest). Після цього виконується системна команда docker push, яка завантажує готові артефакти до публічного або приватного репозиторію.

Завдяки впровадженню описаного конвеєра, процес оновлення програмного забезпечення в цільовому середовищі Kubernetes зводиться до тривіальної зміни тегу образу в конфігураційному маніфесті (Deployment/DaemonSet). Кластер автоматично завантажує нову версію з Docker Hub і виконує плавне оновлення подів (Rolling Update) без переривання процесу моніторингу, що повністю відповідає індустріальним стандартам керування інфраструктурою.

Крім того, автоматизований конвеєр мінімізує ризик людських помилок під час розгортання. У разі нестабільності нової версії експортера, Kubernetes забезпечує миттєвий відкат (Rollback) до попереднього стану.

Інтеграція GitHub Actions із перевірками життєздатності (Probes) гарантує маршрутизацію трафіка лише на успішно запущені контейнери.

Це гарантує максимальну відмовостійкість системи і забезпечує безперервність роботи сервісів (High Availability).

Загальна структурна схема побудованого конвеєра безперервної інтеграції та доставки (CI/CD) наведена на рисунку 4.10.

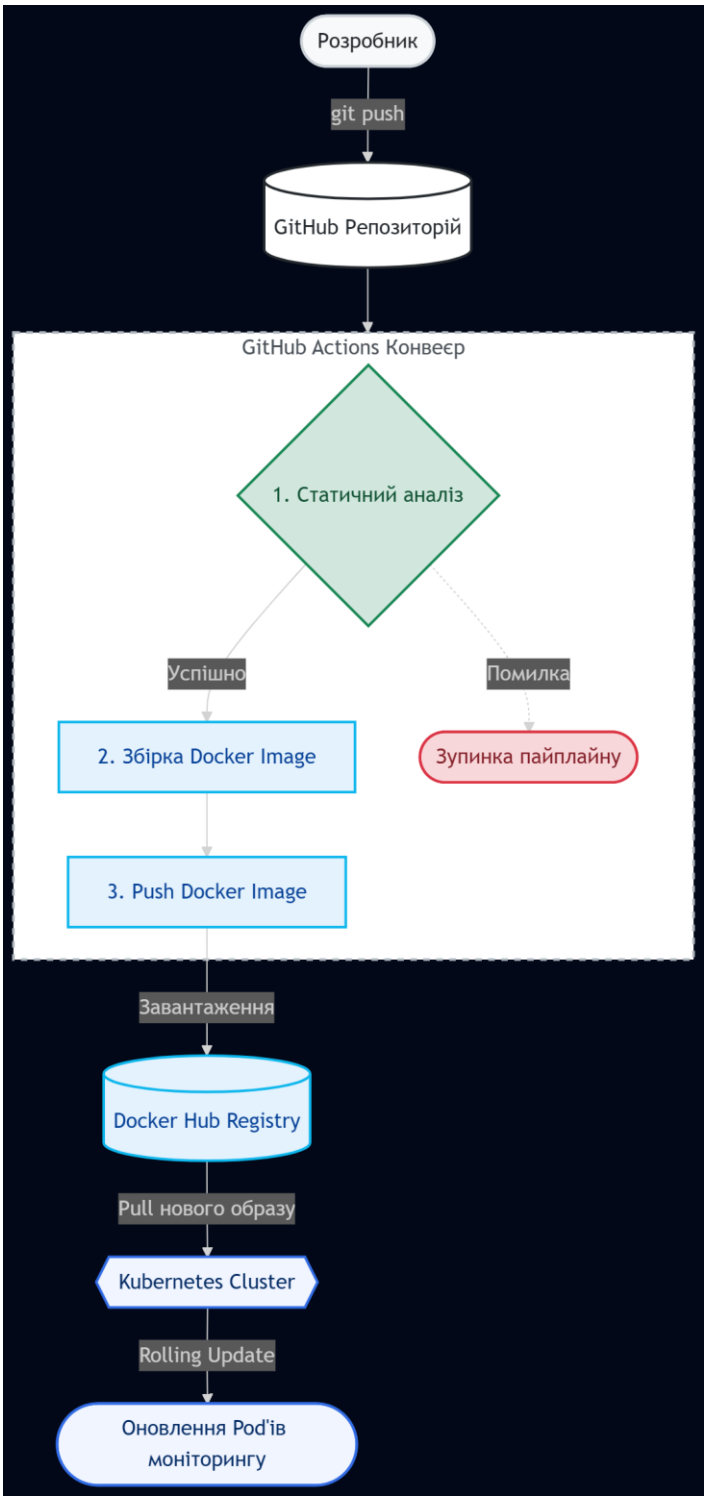


Рисунок 4.10 — Структурна схема конвеєра безперервної інтеграції та доставки (CI/CD)

Впроваджений конвеєр безперервної інтеграції та доставки гарантує високу стабільність коду та автоматизує процес розгортання оновлень системи у цільовому середовищі експлуатації.

ВИСНОВКИ

У дипломній роботі розв’язано актуальну прикладну задачу створення високоефективної системи моніторингу мережевої інфраструктури на базі технології eBPF, що забезпечує глибоку пакетну інспекцію та аналіз телеметрії на транспортному рівні операційної системи Linux без внесення суттєвих накладних витрат.

У ході виконання дипломної роботи було виконано всі поставлені завдання та отримано такі результати:

1) проведено аналіз наявних методів та індустріальних засобів мережевого моніторингу. Доведено, що традиційні підходи не здатні забезпечити достатню деталізацію телеметрії без критичного навантаження на обчислювальні ресурси. Обґрунтовано доцільність застосування технології Extended Berkeley Packet Filter (eBPF) у комбінації з мовою програмування Go для розробки спеціалізованого моніторингового агента;

2) досліджено теоретичні та алгоритмічні основи функціонування мережевого стека Linux. Розроблено подійно-орієнтовані алгоритми безпечного перехоплення ядерних подій (зокрема викликів tcp_retransmit_skb та kfree_skb) за допомогою динамічних зондів kprobes. Реалізовано математичну модель розрахунку коефіцієнта заповненості таблиці відстеження з’єднань (nf_conntrack), що дає можливість превентивно виявляти загрози вичерпання ресурсів і мережеві аномалії;%

3) спроєктовано мікросервісну архітектуру і програмно реалізовано серверну частину (Backend) експортера. Завдяки застосуванню конкурентних патернів мови Go (Goroutines, Channels) та оптимізації роботи з пам’яттю (впровадження пулів об’єктів), досягнуто високої швидкодії обробки даних (Zero-overhead). Впроваджено повну підтримку стандарту OpenMetrics для безперешкодної інтеграції бекенда з базою даних часових рядів Prometheus;

4) розроблено клієнтський вебзастосунок (Frontend) на базі бібліотеки React і строго типізованої мови TypeScript. Створено інтерактивну аналітичну панель

(Dashboard), яка забезпечує візуалізацію багатовимірних часових рядів за допомогою масштабованої векторної графіки (Recharts). Реалізовано механізми динамічної фільтрації даних, вибору часових діапазонів і систему інтернаціоналізації інтерфейсу (i18n);

5) забезпечено повний цикл автоматизації і безпеки розгортання. Програмний комплекс контейнеризовано за стандартами OCI (Docker) з використанням оптимізованої багатоетапної збірки. Спроєктовано маніфести для оркестрації системи в середовищі Kubernetes з суворим дотриманням принципу найменших привілеїв (PoLP) за допомогою гранулярного налаштування Linux Capabilities;

6) налаштовано конвеєр безперервної інтеграції і доставки (CI/CD) на базі платформи GitHub Actions. Це забезпечило можливість автоматизувати процеси статичного аналізу вихідного коду (Linting), безпечної збірки і публікації готових контейнерних образів у віддалені реєстри;

7) проведено експериментальне тестування системи в умовах імітації мережевої деградації за допомогою утиліти Traffic Control. Результати обчислювальних експериментів підтвердили абсолютну коректність захоплення метрик зондами eBPF та їхню миттєву візуалізацію на клієнтському дашборді.

Практична цінність одержаних результатів полягає у створенні готового до промислової експлуатації програмного продукту з відкритим вихідним кодом. Розроблена система здатна стабільно функціонувати в умовах високонавантажених (High-load) та хмарних (Cloud-native) середовищ, надаючи інженерам потужний інструментарій для локалізації вузьких місць мережі, оптимізації продуктивності та підвищення загальної надійності інфраструктури.

Основні положення, архітектурні рішення і практичні результати розробки доповідалися та отримали позитивну оцінку на XXIII Міжнародній науково-практичній конференції молодих вчених та студентів «Сучасні проблеми наукового забезпечення енергетики: безпека, стійкість, IT та екологічний моніторинг» (м. Київ, 17-22 квітня 2026 р) (Додаток Б).

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Малюк І. О., Кублій Л. І. Мережевий експортер метрик вузла у форматі OpenMetrics і його інтеграція в систему моніторингу. Сучасні проблеми наукового забезпечення енергетики: безпека, стійкість, IT та екологічний моніторинг: матеріали XXIII Міжнар. наук.-практ. конф. молодих вчених та студентів, м. Київ, 17-22 квітня 2026 р. Київ: КПІ ім. Ігоря Сікорського, Вид-во «Політехніка», 2026. С. 440–442. URL: https://iate.kpi.ua/uploads/p_182_80374158.pdf.

2. Transmission Control Protocol. IETF RFC 793. URL: https://datatracker.ietf.org/doc/html/rfc793 (дата звернення: 26.05.2026).

3. TCPDUMP & LIBPCAP. The Tcpdump Group. URL: https://www.tcpdump.org/ (дата звернення: 26.05.2026).

4. Zabbix 7.0 Manual. Zabbix Documentation. URL: https://www.zabbix.com/documentation/7.0/en/manual (дата звернення: 26.05.2026).

5. Telegraf Documentation. InfluxData. URL: https://docs.influxdata.com/platform/getting-started/ (дата звернення: 26.05.2026).

6. Network Performance Monitoring. Datadog Docs. URL: https://docs.datadoghq.com/network_monitoring/performance/ (дата звернення: 26.05.2026).

7. Node exporter. Prometheus. URL: https://prometheus.io/docs/guides/node-exporter/ (дата звернення: 26.05.2026).

8. The netfilter.org project. URL: https://netfilter.org/ (дата звернення: 26.05.2026).

9. What is eBPF? An Introduction and Deep Dive. eBPF.io. URL: https://ebpf.io/what-is-ebpf/ (дата звернення: 26.05.2026).

10. The Linux Kernel documentation. Kernel.org. URL: https://docs.kernel.org/ (дата звернення: 26.05.2026).

11. Prometheus — Monitoring system & time series database. URL: https://prometheus.io/ (дата звернення: 26.05.2026).

12. Goroutines. A Tour of Go. URL: https://go.dev/tour/concurrency/1 (дата звернення: 26.05.2026).

13. React – A JavaScript library for building user interfaces. Meta Platforms, Inc. URL: https://react.dev/ (дата звернення: 26.05.2026).

14. TypeScript: Typed JavaScript at Any Scale. Microsoft. URL: https://www.typescriptlang.org/ (дата звернення: 26.05.2026).

15. OpenMetrics Specification. Cloud Native Computing Foundation. URL: https://github.com/OpenObservability/OpenMetrics/blob/main/specification/OpenMetrics.md (дата звернення: 26.05.2026).

16. Recharts. A composable charting library built on React components. URL: https://recharts.org/ (дата звернення: 26.05.2026).

17. A Guide to the Go Garbage Collector. Go Programming Language. URL: https://go.dev/doc/gc-guide (дата звернення: 26.05.2026).

18. Docker Documentation. Docker Inc. URL: https://docs.docker.com/ (дата звернення: 26.05.2026).

19. Kubernetes Documentation. The Kubernetes Authors. URL: https://kubernetes.io/docs/home/ (дата звернення: 26.05.2026).

20. Open Container Initiative Specifications. Linux Foundation. URL: https://opencontainers.org/ (дата звернення: 26.05.2026).

21. Traffic Control HOWTO. Linux Advanced Routing & Traffic Control. URL: https://tldp.org/HOWTO/Traffic-Control-HOWTO/ (дата звернення: 26.05.2026).

22. GitHub Actions Documentation. GitHub Docs. URL: https://docs.github.com/en/actions (дата звернення: 26.05.2026).

ДОДАТОК А

Тези доповіді
у збірнику матеріалів
XXIII Міжнародної науково-практичної конференції молодих вчених та студентів

«Сучасні проблеми наукового забезпечення енергетики: безпека, стійкість, ІТ та екологічний моніторинг»

УКР.НТУУ «КПІ ім. Ігоря Сікорського»_ІАТЕ_ЦТЕ_ТР_21

Аркушів 6

Київ — 2026

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

СУЧАСНІ ПРОБЛЕМИ НАУКОВОГО ЗАБЕЗПЕЧЕННЯ ЕНЕРГЕТИКИ:

БЕЗПЕКА, СТІЙКІСТЬ, ІТ ТА ЕКОЛОГІЧНИЙ МОНІТОРИНГ

XXIII Міжнародна науково-практична конференція
молодих вчених та студентів

До 40-річчя Чорнобильської катастрофи

17–22 квітня 2026 року, м. Київ

Київ-2026

УДК 004.77:005.457

[1] Бакалаврант 4 курсу Малюк І. О.
[1] Доц., к.т.н. Кублій Л. І.
https://scholar.google.com.ua/citations?hl=en&user=rXA1eMIAAAAJ

[1] КПІ ім. Ігоря Сікорського

МЕРЕЖЕВИЙ ЕКСПОРТЕР МЕТРИК ВУЗЛА У ФОРМАТІ OPENMETRICS І ЙОГО ІНТЕГРАЦІЯ В СИСТЕМУ МОНІТОРИНГУ

Постановка проблеми та її актуальність. Сучасні інформаційні системи, вебсервіси й корпоративні застосунки функціонують у середовищах з високим рівнем динамічності – віртуалізована інфраструктура, контейнеризація, хмарні платформи, мікросервісні архітектури. У таких умовах стабільність сервісів значною мірою залежить від стану мережевої підсистеми вузлів: якості мережевих інтерфейсів, коректності роботи TCP/IP-стека, наявності втрат пакетів і помилок передавання, перевантаження каналів зв'язку, а також системних обмежень (наприклад, таблиць з'єднань). Нерідко деградація продуктивності або відмови сервісів викликані саме мережевими аномаліями, які проявляються у вигляді збільшення повторних передач TCP, росту втрат на інтерфейсах, стрибків затримок і нестабільності встановлення з'єднань. Особливо це критично для розподілених систем, де навіть мікродеградація мережі на одному вузлі може призвести до каскадних збоїв у роботі всього застосунку.

Оперативне виявлення і діагностика мережевих проблем потребують регулярного збору метрик з вузлів інфраструктури, їхнього уніфікованого подання і подальшої інтеграції в систему моніторингу з візуалізацією і сповіщеннями. Проте наявні універсальні агенти або експортери метрик часто надають тільки базовий набір показників або не забезпечують достатню деталізацію мережевих характеристик, необхідну для практичних сценаріїв DevOps/SRE (аналіз інцидентів, виявлення деградації, контроль якості мережевих з'єднань, оцінка пропускної здатності). Це обумовлює необхідність створення спеціалізованого мережевого експортера, орієнтованого на розширені мережеві метрики і сумісного зі стандартним форматом експорту. Стандарт OpenMetrics забезпечує узгоджений підхід до опису й передачі метрик, що спрощує інтеграцію з інструментами збору й аналізу часових рядів даних, а також сприяє відтворюваності й масштабованості моніторингу. Таким чином, актуальним є створення рішення, яке збирає розширені мережеві метрики вузла, експортує їх у форматі OpenMetrics та інтегрується з типовим контуром моніторингу (збір – зберігання – інформаційні панелі – сповіщення).

Аналіз останніх досліджень. Побудова систем моніторингу інфраструктури здебільшого ґрунтується на принципах спостережуваності, де метрики є одним із ключових сигналів поряд із лог-файлами і відстеженням маршрутів. У практиці експлуатації широко застосовують модель опитування або підхід витягування (pull-модель), за якої система моніторингу періодично запитує кінцеву точку (endpoint) метрик вузла. Стандарт OpenMetrics визначає формат подання метрик, їхні типи (лічильники, значення стану), правила іменування й метадані, що забезпечує сумісність між компонентами екосистеми моніторингу [1].

Для збору мережевої статистики в ОС Linux традиційно використовуються системні джерела, зокрема файли підсистеми procfs (наприклад, статистика інтерфейсів і мережевих протоколів), а також утиліти для перегляду станів програмних інтерфейсів обміну даними між процесами (сокетів) і лічильників TCP/IP-стека [2]. На основі таких даних можуть будуватися метрики пропускної здатності, помилок і втрат на інтерфейсах, повторних передач TCP, показників встановлення чи закриття з'єднань тощо [3]. У практичних рішеннях важливо також контролювати кардинальність метрик (щоб уникнути надмірної

кількості комбінацій міток), тому розробники часто обмежуються агрегованими метриками на рівні інтерфейсу або вузла [2]. У сучасних інфраструктурних середовищах візуалізація й аналіз метрик часто реалізуються через інформаційні панелі (дашборди), які відображають ключові показники трафіку й помилок, а також через правила сповіщення, які дають можливість автоматично повідомляти інженерів про аномалії [4]. Відтак, актуальним є створення спеціалізованого експортера метрик, який доповнює стандартний набір показників необхідними мережевими індикаторами і легко інтегрується в типовий моніторинговий стек.

Формулювання мети. Метою роботи є створення системи збору та експорту розширених мережевих метрик вузла у форматі OpenMetrics та інтеграція її в систему моніторингу. Система забезпечує: збір статистики мережевих інтерфейсів і TCP/IP-стека на вузлі; формування метрик у форматі OpenMetrics і надання HTTP-кінцевої точки для опитування; інтеграцію з системою моніторингу й візуалізації; налаштування сповіщень про мережеві аномалії на основі правил; оцінку продуктивності й накладних витрат роботи експортера.

Основна частина. Для досягнення поставленої мети:

– визначено перелік інформативних мережевих метрик для експлуатаційних сценаріїв (загальний трафік, пакети, апаратні помилки, втрати, повторні передавання TCP, помилки встановлення з'єднань, агреговані показники станів сокетів);

– спроєктовано модель метрик (імена, типи, одиниці вимірювання, рекомендовані мітки) із врахуванням обмеження кардинальності;

– реалізовано модуль збору даних із системних джерел ОС Linux (зокрема, структурування файлів підсистеми procfs і використання інших системних інтерфейсів);

– реалізовано HTTP-сервер, що надає кінцеву точку /metrics, яка генерує дані в сумісному OpenMetrics-форматі;

– забезпечено гнучку конфігурацію (частота оновлення, список інтерфейсів, мережеві пороги для сповіщень, параметри роботи сервісу);

– реалізовано контейнеризацію і відтворюване розгортання в тестовому середовищі за допомогою платформи Docker;

– налаштовано контур інтеграції: збір метрик, інформаційні панелі, сповіщення;

– проведено експериментальну оцінку таких показників, як стабільність, вплив на процесор та оперативну пам'ять, час відповіді кінцевої точки, коректність при різних інтервалах опитування.

При створенні системи обрано сучасні галузеві інструменти. Як основний засіб реалізації програмної частини експортера використано мову програмування Go (Golang). Вибір цієї мови обґрунтований її високою продуктивністю, низьким споживанням системних ресурсів (що є критичним для агентів моніторингу на кінцевих вузлах), а також потужними вбудованими засобами для конкурентного виконання задач. Крім того, екосистема Go має офіційну й найбільш стабільну бібліотеку для роботи з метриками (Prometheus client library), що значно спрощує процес нормалізації даних у формат OpenMetrics і розгортання HTTP-сервера. Завдяки статичній компіляції, готовий експортер розповсюджується у вигляді єдиного бінарного файлу без зовнішніх залежностей, що ідеально підходить для мікросервісних і контейнеризованих середовищ [5]. Для збору часових рядів даних обрано систему Prometheus [6], оскільки вона нативно використовує підхід витягування (pull) для збору даних і повністю підтримує формат OpenMetrics. Для візуалізації та сповіщень застосовано платформу Grafana [4], яка тісно інтегрується із системою Prometheus. Для отримання системних метрик обрано підсистему procfs (файли /proc/net/dev і /proc/net/snmp), оскільки цей метод є найшвидшим і не створює надмірного навантаження на систему, на відміну від перехоплення пакетів на рівні ядра.

Запропонована система складається з мережевого експортера метрик, який виконується на вузлі як окремий сервіс і надає HTTP-кінцеву точку /metrics. Збір метрик організовано циклічно – із заданим інтервалом експортер зчитує системні лічильники,

виконує нормалізацію значень та оновлює відповідні метрики у своїй пам'яті. Далі зовнішня система моніторингу періодично опитує цю точку і зберігає часові ряди даних для подальшого аналізу. У системі передбачено кілька груп метрик:

– метрики інтерфейсів – для кожного мережевого інтерфейсу формуються лічильники прийнятих/переданих байтів і пакетів, а також лічильники помилок і втрат пакетів. На основі цих даних розраховується швидкість трафіку, відсоток помилок, виявляються проблемні інтерфейси;

– метрики TCP/IP-стека – формуються агреговані показники роботи TCP такі, як кількість переданих/отриманих сегментів, повторні передавання, невдалі спроби встановлення з'єднань, скидання. Ці метрики є особливо корисними для діагностики проблем, коли втрати пакетів або затримки неочевидні на рівні фізичного інтерфейсу, але проявляються на логічному рівні TCP;

– агреговані метрики станів з'єднань – для зменшення кардинальності система не збирає детальну статистику щодо кожного порту за кожною IP-адресою, а формує узагальнені показники (наприклад, загальна кількість з'єднань у стані ESTABLISHED або TIME_WAIT). Це забезпечує практичну цінність для експлуатації без надмірного навантаження на сховище.

Важливою частиною є інтеграція з контуром моніторингу. Налаштовується автоматичний збір метрик, створюються інформаційні панелі, які відображають ключові графіки швидкості трафіку, динаміку помилок і втрат по інтерфейсах, показники повторних передавань TCP і загальну кількість активних з'єднань. Додатково створюються жорсткі правила сповіщень для виявлення таких аномалій, як стале зростання втрат або помилок на інтерфейсі, зростання TCP-ретрансмісій, різке падіння чи зростання обсягів трафіку, недоступність кінцевої точки експортера. Для демонстрації працездатності передбачено сценарії імітації навантаження і погіршення мережевих умов у тестовому середовищі з подальшим аналізом графіків і спрацювання сповіщень. Виконується оцінка накладних витрат експортера (CPU/RAM) залежно від інтервалу оновлення й частоти опитування.

Висновки. Запропонована система забезпечує збір розширених мережевих метрик вузла і надає їх у форматі OpenMetrics, що спрощує інтеграцію з сучасними інструментами моніторингу. Реалізація експортера та налаштування візуалізації і сповіщень дають можливість оперативно виявляти мережеві аномалії, підвищувати прозорість стану мережевої підсистеми і значно скорочувати час діагностики інцидентів. Подальший розвиток роботи може включати розширення набору метрик (наприклад, показники підсистеми відстеження з'єднань conntrack), підтримку вибіркового збору для контейнеризованих середовищ, а також поглиблену інтеграцію з розподіленими сценаріями моніторингу в кластерній інфраструктурі.

Перелік посилань:

1. OpenMetrics. Specification : вебсайт. URL: https://github.com/OpenObservability/OpenMetrics/blob/main/specification/OpenMetrics.md (дата звернення: 12.02.2026).

2. Kerrisk M. The Linux Programming Interface: A Linux and UNIX System Programming Handbook. San Francisco: No Starch Press, 2010. 1552 p. URL: https://broman.dev/download/The%20Linux%20Programming%20Interface.pdf (дата звернення: 21.02.2026).

3. Transmission Control Protocol (TCP). RFC 9293. August 2022. URL: https://datatracker.ietf.org/doc/html/rfc9293 (дата звернення: 13.02.2026).

4. Grafana Documentation : вебсайт. URL: https://grafana.com/docs/grafana/latest/ (дата звернення: 13.02.2026).

5. The Go Programming Language : вебсайт. URL: https://go.dev/ (дата звернення: 21.02.2026).

6. Prometheus Documentation: Metric types : вебсайт. URL: https://prometheus.io/docs/concepts/metric_types/ (дата звернення: 13.02.2026).

ДОДАТОК Б

Фрагменти програмного коду розробленого комплексу

«Мережевий експортер метрик вузла у форматі OpenMetrics і його інтеграція в систему моніторингу»

УКР.НТУУ «КПІ ім. Ігоря Сікорського»_ІАТЕ_ЦТЕ_ТР_21

Мови програмування — Go, C, TypeScript

Аркушів 14

Київ — 2026

Лістинг програмного коду розробленого комплексу

Програмні засоби:

— Мова програмування для бекенда – Go (Golang);

— Технологія низькорівневого трасування – eBPF (Extended Berkeley Packet Filter) із використанням мови C;

— Бібліотека для взаємодії з eBPF – Cilium eBPF (bpf2go);

— Стандарт представлення метрик – OpenMetrics;

— Система моніторингу та часова база даних – Prometheus (TSDB);

— Мова програмування для фронтенда – TypeScript (React);

— Бібліотека візуалізації даних – Recharts;

— Система оркестрації та контейнеризації – Kubernetes, Docker.

Лістинг програмного коду розробленого комплексу

```
============================================================
ЧАСТИНА 1. НИЗЬКОРІВНЕВІ КОМПОНЕНТИ ЯДРА (eBPF C)
============================================================

Файл: bpf/tcp_retransmit.bpf.c
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>
#include <bpf/bpf_core_read.h>

struct event_t {
    __u32 saddr;
    __u32 daddr;
    __u16 dport;
    __u16 type; // 1 = Retransmit, 2 = Drop
    __u64 ts;
};

struct {
    __uint(type, BPF_MAP_TYPE_RINGBUF);
    __uint(max_entries, 1 << 24);
} events SEC(".maps");

SEC("kprobe/tcp_retransmit_skb")
int handle_retransmit(struct pt_regs *ctx)
{
```

```
    struct sock *sk = (struct sock *)PT_REGS_PARM1(ctx);
    if (!sk)
        return 0;

    struct event_t *e;
    e = bpf_ringbuf_reserve(&events, sizeof(*e), 0);
    if (!e)
        return 0;

    e->saddr = BPF_CORE_READ(sk, __sk_common.skc_rcv_saddr);
    e->daddr = BPF_CORE_READ(sk, __sk_common.skc_daddr);
    e->dport = BPF_CORE_READ(sk, __sk_common.skc_dport);
    e->type = 1;
    e->ts = bpf_ktime_get_ns();

    bpf_ringbuf_submit(e, 0);
    return 0;
}

SEC("tp/skb/kfree_skb")
int handle_drop(struct trace_event_raw_kfree_skb *ctx)
{
    struct sk_buff *skb = (struct sk_buff *)ctx->skbaddr;
    struct sock *sk = BPF_CORE_READ(skb, sk);

    if (!sk)
        return 0;

    struct event_t *e;
    e = bpf_ringbuf_reserve(&events, sizeof(*e), 0);
    if (!e)
        return 0;

    e->saddr = BPF_CORE_READ(sk, __sk_common.skc_rcv_saddr);
    e->daddr = BPF_CORE_READ(sk, __sk_common.skc_daddr);
    e->dport = BPF_CORE_READ(sk, __sk_common.skc_dport);
    e->type = 2;
    e->ts = bpf_ktime_get_ns();

    bpf_ringbuf_submit(e, 0);
    return 0;
}

char LICENSE[] SEC("license") = "GPL";


============================================================
ЧАСТИНА 2. СЕРВЕРНИЙ КОМПОНЕНТ МОНІТОРИНГУ (Go Backend)
============================================================

Файл: cmd/main.go
package main
```

```
import (
    "context"
    "errors"
    "log"
    "log/slog"
    "net/http"
    "network-exporter/internal/adapters/ebpf"
    "os"
    "os/signal"
    "syscall"
    "time"

    "network-exporter/internal/adapters/procfs"
    "network-exporter/internal/application/collectors"
    "network-exporter/internal/ebpf/gen"

    "github.com/cilium/ebpf/link"
    "github.com/cilium/ebpf/ringbuf"
    "github.com/cilium/ebpf/rlimit"
    "github.com/prometheus/client_golang/prometheus"
    "github.com/prometheus/client_golang/prometheus/promhttp"
)

func main() {
    ctx, stop := signal.NotifyContext(context.Background(), syscall.SIGINT, syscall.SIGTERM)
    defer stop()

    logger := slog.New(slog.NewJSONHandler(os.Stdout, nil))
    slog.SetDefault(logger)
    slog.Info("Application started")

    if err := rlimit.RemoveMemlock(); err != nil {
        log.Fatalf("failed to remove memlock: %v", err)
    }

    objs := gen.TcpRetransmitObjects{}
    if err := gen.LoadTcpRetransmitObjects(&objs, nil); err != nil {
        log.Fatalf("loading objects: %v", err)
    }
    defer objs.Close() //nolint:errcheck

    kp, err := link.Kprobe("tcp_retransmit_skb", objs.HandleRetransmit, nil)
    if err != nil {
        log.Fatalf("failed to attach kprobe: %v", err)
    }
    defer kp.Close() //nolint:errcheck

    tp, err := link.Tracepoint("skb", "kfree_skb", objs.HandleDrop, nil)
    if err != nil {
        log.Fatalf("failed to attach tracepoint: %v", err)
    }
```

```
defer tp.Close() //nolint:errcheck

rd, err := ringbuf.NewReader(objs.Events)
if err != nil {
    log.Fatalf("opening ringbuf reader: %v", err)
}
defer rd.Close() //nolint:errcheck

fs := procfs.New()
networkCollector := collectors.NewNetworkCollector(fs)
tcpCollector := collectors.NewTCPCollector(fs)
socketCollector := collectors.NewSocketCollector(fs)
retransmitColl := collectors.NewRetransmitCollector()
conntrackCollector := collectors.NewConntrackCollector(fs)
udpCollector := collectors.NewUDPCollector(fs)
icmpCollector := collectors.NewICMPCollector(fs)

ebpfExporter := ebpf.NewRetransmitExporter(retransmitColl, 4, 2048)
ebpfExporter.Start(ctx)

reg := prometheus.NewRegistry()
reg.MustRegister(
    networkCollector,
    tcpCollector,
    socketCollector,
    retransmitColl,
    conntrackCollector,
    udpCollector,
    icmpCollector,
)

go func() {
    slog.Info("eBPF Producer started")
    for {
        record, err := rd.Read()
        if err != nil {
            if errors.Is(err, ringbuf.ErrClosed) {
                return
            }
            continue
        }
        ebpfExporter.Publish(record.RawSample)
    }
}()

mux := http.NewServeMux()
mux.Handle("/metrics", promhttp.HandlerFor(reg, promhttp.HandlerOpts{}))

server := &http.Server{
    Addr:        ":8080",
    Handler:     mux,
    ReadTimeout:  5 * time.Second,
```

```
        WriteTimeout: 10 * time.Second,
    }

    go func() {
        slog.Info("Server starting on port 8080")
        if err := server.ListenAndServe(); err != nil && !errors.Is(err, http.ErrServerClosed) {
            log.Fatalf("listen: %s\n", err)
        }
    }()

    <-ctx.Done()
    slog.Info("shutting down server...")

    shutdownCtx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
    defer cancel()
    if err := server.Shutdown(shutdownCtx); err != nil {
        log.Printf("HTTP server shutdown error: %v", err)
    }

    slog.Info("server shutdown successfully")
}
```

Файл: internal/adapters/ebpf/retransmit_exporter.go

```
package ebpf

import (
    "bytes"
    "context"
    "encoding/binary"
    "log/slog"
    "net"
    "network-exporter/internal/application/collectors"
    "sync"
)

type RetransmitEvent struct {
    Saddr [4]byte
    Daddr [4]byte
    Dport uint16
    Type  uint16 // 1 = Retransmit, 2 = Drop
    _     [4]byte
    Ts    uint64
}

type Exporter struct {
    collector  *collectors.RetransmitCollector
    workers    int
    bufferSize int
    eventChan  chan []byte
    wg         sync.WaitGroup
}
```

```
func NewRetransmitExporter(c *collectors.RetransmitCollector, workers, buffer int) *Exporter {
    return &Exporter{
        collector:  c,
        workers:    workers,
        bufferSize: buffer,
        eventChan:  make(chan []byte, buffer),
    }
}

func (e *Exporter) Start(ctx context.Context) {
    slog.Info("Starting eBPF worker pool", "workers", e.workers, "buffer", e.bufferSize)
    for i := 0; i < e.workers; i++ {
        e.wg.Add(1)
        go e.worker(ctx)
    }
}

func (e *Exporter) worker(ctx context.Context) {
    defer e.wg.Done()
    for {
        select {
        case <-ctx.Done():
            return
        case raw, ok := <-e.eventChan:
            if !ok {
                return
            }
            e.processEvent(raw)
        }
    }
}

func (e *Exporter) processEvent(raw []byte) {
    slog.Info("Processing event", "raw", string(raw))
    var event RetransmitEvent
    if err := binary.Read(bytes.NewBuffer(raw), binary.LittleEndian, &event); err != nil {
        slog.Error("Failed to parse event", "error", err)
        return
    }

    srcIP := net.IP(event.Saddr[:]).String()
    dstIP := net.IP(event.Daddr[:]).String()
    dport := binary.BigEndian.Uint16(binary.LittleEndian.AppendUint16(nil, event.Dport))

    e.collector.Observe(srcIP, dstIP, dport, event.Type)
}

func (e *Exporter) Publish(raw []byte) {
    select {
    case e.eventChan <- raw:
    default:
```

```
            slog.Debug("Event dropped due to backpressure")
        }
}

func (e *Exporter) Wait() {
    close(e.eventChan)
    e.wg.Wait()
}
```

Файл: internal/application/collectors/ebpf_common.go

```
package collectors

import (
    "strconv"

    "github.com/prometheus/client_golang/prometheus"
)

type RetransmitCollector struct {
    retransmits *prometheus.CounterVec
    drops       *prometheus.CounterVec
}

func NewRetransmitCollector() *RetransmitCollector {
    return &RetransmitCollector{
        retransmits: prometheus.NewCounterVec(
            prometheus.CounterOpts{
                Name: "node_tcp_retransmit_total",
                Help: "Total number of TCP retransmits detected via eBPF",
            },
            []string{"src_ip", "dst_ip", "dport"},
        ),
        drops: prometheus.NewCounterVec(
            prometheus.CounterOpts{
                Name: "node_tcp_drops_total",
                Help: "Total number of TCP drops detected via eBPF",
            },
            []string{"src_ip", "dst_ip", "dport"},
        ),
    }
}

func (c *RetransmitCollector) Describe(ch chan<- *prometheus.Desc) {
    c.retransmits.Describe(ch)
    c.drops.Describe(ch)
}

func (c *RetransmitCollector) Collect(ch chan<- prometheus.Metric) {
    c.retransmits.Collect(ch)
    c.drops.Collect(ch)
}
```

```
func (c *RetransmitCollector) Observe(src, dst string, dport uint16, eventType uint16) {
    portStr := strconv.Itoa(int(dport))

    switch eventType {
    case 1:
        c.retransmits.WithLabelValues(src, dst, portStr).Inc()
    case 2:
        c.drops.WithLabelValues(src, dst, portStr).Inc()
    }
}
```

```
================================================================
ЧАСТИНА 3. АДАПТЕРИ ЗБОРУ МЕРЕЖЕВОЇ СТАТИСТИКИ (ProcFS Parser)
================================================================
```

Файл: internal/adapters/procfs/procfs.go

```
package procfs

type ProcFS struct{}

func New() *ProcFS {
    return &ProcFS{}
}
```

Файл: internal/adapters/procfs/conntrack.go

```
package procfs

import (
    "network-exporter/internal/domain/models"
    "os"
    "strconv"
    "strings"
)

func (p *ProcFS) ReadConntrackStats() (models.ConntrackStats, error) {
    countStr, err := os.ReadFile("/proc/sys/net/netfilter/nf_conntrack_count")
    if err != nil {
        return models.ConntrackStats{}, err
    }

    maxStr, err := os.ReadFile("/proc/sys/net/netfilter/nf_conntrack_max")
    if err != nil {
        return models.ConntrackStats{}, err
    }

    count, _ := strconv.ParseUint(strings.TrimSpace(string(countStr)), 10, 64)
    max, _ := strconv.ParseUint(strings.TrimSpace(string(maxStr)), 10, 64)

    return models.ConntrackStats{
```

```
            Count: count,
            Max:   max,
        }, nil
}


Файл: internal/adapters/procfs/tcp.go
package procfs

import (
    "bufio"
    "network-exporter/internal/domain/models"
    "os"
    "strconv"
    "strings"
)

func (p *ProcFS) ReadTCPStats() (models.TCPStats, error) {
    f, err := os.Open("/proc/net/snmp")
    if err != nil {
        return models.TCPStats{}, err
    }
    defer f.Close() //nolint:errcheck

    scanner := bufio.NewScanner(f)
    var keys []string
    var values []string

    for scanner.Scan() {
        line := scanner.Text()
        if strings.HasPrefix(line, "Tcp:") {
            parts := strings.Fields(line)
            if keys == nil {
                keys = parts[1:]
            } else {
                values = parts[1:]
            }
        }
    }

    var parse = func(name string) uint64 {
        for i, k := range keys {
            if k == name && i < len(values) {
                v, _ := strconv.ParseUint(values[i], 10, 64)
                return v
            }
        }
        return 0
    }

    return models.TCPStats{
        ActiveOpens:  parse("ActiveOpens"),
```

```
            PassiveOpens: parse("PassiveOpens"),
            AttemptFails: parse("AttemptFails"),
            EstabResets:  parse("EstabResets"),
            InSegs:       parse("InSegs"),
            OutSegs:      parse("OutSegs"),
            RetransSegs:  parse("RetransSegs"),
        }, nil
}
```

Файл: internal/application/collectors/conntrack_collector.go

```
package collectors

import (
    "github.com/prometheus/client_golang/prometheus"
    "network-exporter/internal/ports"
)

type ConntrackCollector struct {
    reader ports.ConntrackReader
    count  *prometheus.Desc
    max    *prometheus.Desc
}

func NewConntrackCollector(r ports.ConntrackReader) *ConntrackCollector {
    return &ConntrackCollector{
        reader: r,
        count: prometheus.NewDesc(
            "node_network_conntrack_count",
            "Number of allocated connection tracking entries",
            nil, nil,
        ),
        max: prometheus.NewDesc(
            "node_network_conntrack_max",
            "Maximum number of connection tracking entries",
            nil, nil,
        ),
    }
}

func (c *ConntrackCollector) Describe(ch chan<- *prometheus.Desc) {
    ch <- c.count
    ch <- c.max
}

func (c *ConntrackCollector) Collect(ch chan<- prometheus.Metric) {
    stats, err := c.reader.ReadConntrackStats()
    if err != nil {
        return
    }

    ch <- prometheus.MustNewConstMetric(c.count, prometheus.GaugeValue, float64(stats.Count))
```

```
    ch <- prometheus.MustNewConstMetric(c.max, prometheus.GaugeValue, float64(stats.Max))
}
```

```
============================================================
ЧАСТИНА 4. КЛІЄНТСЬКИЙ ВЕБ-ІНТЕРФЕЙС (TypeScript & React)
============================================================
```

```
Файл: frontend/src/hooks/usePrometheus.ts
import { useState, useEffect, useCallback, useRef } from 'react';
import { queryRange, formatTimeSeriesData, buildZeroSkeleton, SCRAPE_INTERVAL } from '../lib/prometheus';
import { useDashboard } from '../context/DashboardContext';

interface UsePrometheusOptions {
  query: string;
  stepSeconds?: number;
  metricNameKey?: string;
  skeletonKeys?: string[];
  fallbackSeriesName?: string;
}

export const usePrometheus = ({
  query,
  stepSeconds = SCRAPE_INTERVAL,
  metricNameKey,
  skeletonKeys = [],
  fallbackSeriesName = 'Series 1',
}: UsePrometheusOptions) => {
  const { getEffectiveWindow, refreshTick, timeRangeMinutes, absoluteRange, frozenRange } = useDashboard();

  const isFirstLoad = useRef(true);
  const [data, setData] = useState<any[]>([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState<string | null>(null);
  const prevDataRef = useRef<string>('[]');

  const fetchData = useCallback(async () => {
    try {
      const window = getEffectiveWindow();
      const end = Math.floor(window.end / 5) * 5;
      const start = end - (window.end - window.start);
      const step = 5;

      const results = await queryRange(query, start, end, step);

      const newData =
        results.length === 0
          ? buildZeroSkeleton(start, end, stepSeconds, skeletonKeys)
          : formatTimeSeriesData(results, metricNameKey, fallbackSeriesName);

      const newDataStr = JSON.stringify(newData);
      if (newDataStr !== prevDataRef.current) {
```

```
        prevDataRef.current = newDataStr;
        setData(newData);
       }
       setError(null);
     } catch (err: any) {
       setError(err.message || 'Failed to fetch data');
     } finally {
       if (isFirstLoad.current) {
         isFirstLoad.current = false;
         setLoading(false);
       }
     }
   }, [query, stepSeconds, metricNameKey, fallbackSeriesName, JSON.stringify(skeletonKeys),
getEffectiveWindow]);

   useEffect(() => {
     isFirstLoad.current = true;
     prevDataRef.current = '[]';
     setLoading(true);
     fetchData();
   }, [timeRangeMinutes, absoluteRange, frozenRange, fetchData]);

   useEffect(() => {
     if (refreshTick === 0) return;
     fetchData();
   }, [refreshTick, fetchData]);

   return { data, loading, error, refetch: fetchData };
};

Файл frontend/src/i18n/config.ts
import i18n from 'i18next';
import { initReactI18next } from 'react-i18next';
import enTranslations from './locales/en.json';
import ukTranslations from './locales/uk.json';

i18n
.use(initReactI18next)
.init({
resources: {
en: { translation: enTranslations },
uk: { translation: ukTranslations }
},
lng: 'en',
fallbackLng: 'en',
interpolation: {
escapeValue: false
}
});

export default i18n;
```

Файл frontend/src/components/LanguageSwitcher.tsx

```
import { useTranslation } from 'react-i18next';
import { Globe } from 'lucide-react';

export const LanguageSwitcher = () => {
const { i18n } = useTranslation();

const toggleLanguage = () => {
const nextLang = i18n.language === 'en' ? 'uk' : 'en';
i18n.changeLanguage(nextLang);
};

return (
<button
onClick={toggleLanguage}
className="flex items-center gap-2 px-3 py-2 text-sm font-medium transition-colors rounded-lg bg-cardHover text-textPrimary hover:bg-card border border-white/10"
>
<Globe size={16} />
<span>{i18n.language.toUpperCase()}</span>
</button>
);
};
```

==========

ЧАСТИНА 5. ІНФРАСТРУКТУРА, ОРКЕСТРАЦІЯ ТА CI/CD (YAML)

==========

Файл: .github/workflows/workflow.yml

```
name: CI/CD Pipeline
on:
  push:
    branches: [ "master" ]
jobs:
  build-and-push:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repository
        uses: actions/checkout@v3

      - name: Set up Go
        uses: actions/setup-go@v5
        with:
          go-version: '1.26'

      - name: Install eBPF build dependencies
        run: |
          sudo apt-get update
          sudo apt-get install -y clang llvm libbpf-dev linux-libc-dev gcc-multilib

      - name: Generate eBPF code
        run: go generate ./...
```

```
      - name: Run golangci-lint
        uses: golangci/golangci-lint-action@v7
        with:
          version: v2.12.2

      - name: Login to Docker Hub
        uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}

      - name: Build and push Docker image
        uses: docker/build-push-action@v4
        with:
          context: .
          file: ./build/Dockerfile
          push: true
          tags: ${{ secrets.DOCKERHUB_USERNAME }}/ebpf-exporter:latest
```

Файл: deploy/k8s/daemonset.yaml

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: network-exporter
  labels:
    app: network-exporter
spec:
  selector:
    matchLabels:
      app: network-exporter
  template:
    metadata:
      labels:
        app: network-exporter
    spec:
      hostNetwork: true
      hostPID: true
      containers:
        - name: exporter
          image: network-exporter-ebpf:latest
          imagePullPolicy: IfNotPresent
          volumeMounts:
            - name: tracefs
              mountPath: /sys/kernel/tracing
              readOnly: true
          ports:
            - containerPort: 8080
              name: metrics
          resources:
            requests:
```

```
        cpu: 50m
        memory: 64Mi
      limits:
        cpu: 200m
        memory: 128Mi
    readinessProbe:
      httpGet:
        path: /metrics
        port: 8080
      initialDelaySeconds: 3
      periodSeconds: 10
    livenessProbe:
      httpGet:
        path: /metrics
        port: 8080
      initialDelaySeconds: 10
      periodSeconds: 20
    securityContext:
      privileged: true
volumes:
  - name: tracefs
    hostPath:
      path: /sys/kernel/tracing
      type: Directory
```