МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Західноукраїнський національний університет
Факультет комп'ютерних інформаційних технологій
Кафедра інформаційно-обчислювальних систем і управління

ЦИМБАЛЮК Антон Миколайович

Програмний модуль моніторингу на базі eBPF для підвищення мережевої продуктивності Linux / An eBPF-based monitoring module for Linux network performance optimization

Спеціальність 122 – Комп'ютерні науки
Освітньо-професійна програма – Комп'ютерні науки

Кваліфікаційна робота

Виконав студент групи КН-41
А. М. Цимбалюк

Науковий керівник:
к.т.н., доцент, О.Р. Осолінський

Кваліфікаційну роботу допущено до захисту
«___»_____ 2026 р.

Завідувач кафедри
_____ Н. В. Дзюбановська

Тернопіль – 2026

Факультет комп'ютерних інформаційних технологій
Кафедра інформаційно-обчислювальних систем і управління
Освітній ступінь «бакалавр»
спеціальність 122 – Комп'ютерні науки
освітньо-професійна програма – Комп'ютерні науки

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ Н.В. Дзюбановська
«____» _____20 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ
ЦИМБАЛЮКУ Антону Миколайовичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи: Програмний модуль моніторингу на базі eBPF для підвищення мережевої продуктивності Linux / An eBPF-based monitoring module for Linux network performance optimization
керівник роботи О.Р. Осолінський, к.т.н., доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від 01 грудня 2025 року № 894.
2. Строк подання студентом закінченої кваліфікаційної роботи 25 травня 2026 р.
3. Вихідні дані до кваліфікаційної роботи: завдання на кваліфікаційну роботу студента, наукові статті, технічна література.
4. Основні питання, які потрібно розробити:
- провести аналіз методів формування мережевої телеметрії та телеметрії потоків у Linux;
- провести огляд і аналіз інструментів моніторингу мережевого стеку Linux;
- сформувати постановку задачі;
- розробити загальну структуру проектованого модуля;
- розробити алгоритмічне забезпечення модуля;
- розробити інформаційне забезпечення модуля;
- реалізувати програмне забезпечення модуля;
- розробити інтерфейс взаємодії з користувачем та засоби подання результатів;
- провести тестування та аналіз результатів роботи модуля.

5. Перелік графічного матеріалу в роботі:
- архітектура модуля;
- блок-схема алгоритму роботи модуля моніторингу на базі eBPF.

6. Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 01 грудня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів кваліфікаційної роботи	Примітка
1	Огляд літературних джерел відповідно до предметної області та складання плану роботи	до 01.01.2026 р.	
2	Написання 1 розділу кваліфікаційної роботи	до 01.02.2026 р.	
3	Написання 2 розділу кваліфікаційної роботи	до 15.03.2026 р.	
4	Написання 3 розділу кваліфікаційної роботи	до 01.05.2026 р.	
5	Представлення попереднього варіанту кваліфікаційної роботи, перевірка та внесення змін керівником	до 15.05.2026 р.	
6	Опрацювання зауважень та представлення завершеного варіанту кваліфікаційної роботи. Підготовка супроводжуючих документів	до 25.05.2026 р.	
7	Перевірка кваліфікаційної роботи на оригінальність тексту	до 30.05.2026 р.	
8	Оформлення кваліфікаційної роботи та отримання допуску до захисту	до 10.06.2026 р.	
9	Подання кваліфікаційної роботи до захисту на засіданні атестаційної комісії	до 10.06.2026 р.	

Студент_____ А. М. Цимбалюк
(підпис) (прізвище та ініціали)

Керівник кваліфікаційної роботи_____ О.Р. Осолінський
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Кваліфікаційна робота на тему «Програмний модуль моніторингу на базі eBPF для підвищення мережевої продуктивності Linux» на здобуття освітнього ступеня «бакалавр» з спеціальності 122 «Комп'ютерні науки» освітньо-професійної програми «Комп'ютерні науки» містить 80 сторінок, 23 ілюстрації, 4 додатки та 31 використаних джерел.

Метою кваліфікаційної роботи є розроблення програмного модуля моніторингу на базі eBPF, який забезпечує спостережуваність мережевого стеку Linux для подальшого підвищення мережевої продуктивності за рахунок збору, агрегації та подання телеметрії у зручному для аналізу вигляді.

В роботі використано методи структурно-функціонального аналізу, логіко-алгоритмічного моделювання, порівняльного аналізу інструментів моніторингу, а також технології eBPF, засоби експорту метрик

В роботі проаналізовано мережеву телеметрію в Linux, та сучасні інструменти спостереження за мережевим стеком. Запропоновано архітектуру модуля, побудовану за дворівневим принципом. Реалізовано збір TCP retransmit-подій, механізм передавання подій.

Практична цінність роботи полягає в тому, що реалізований модуль дозволяє виявляти події TCP retransmit у ядрі Linux, виконувати їх інтервальну агрегацію та інтегрувати результати для різних аналітичних систем.

Ключові слова: eBPF, LINUX, МЕРЕЖЕВИЙ СТЕК, МЕРЕЖЕВА ТЕЛЕМЕТРІЯ, FLOW-ТЕЛЕМЕТРІЯ, TCP RETRANSMIT, PROMETHEUS, DASHBOARD, JSON SNAPSHOTS, МОНІТОРИНГ.

ANNOTATION

Explanatory note to the qualification thesis: 80 pages, 23 figures, 31 references, 4 annexes.

The aim of this work is to develop an eBPF-based monitoring software module that provides observability of the Linux network stack in order to further improve network performance through the collection, aggregation, and presentation of telemetry in a form convenient for analysis.

The work employs methods of structural and functional analysis, logical and algorithmic modeling, comparative analysis of monitoring tools, as well as eBPF technologies and metric export tools.

The thesis analyzes network telemetry in Linux and modern tools for observing the network stack. A two-level module architecture is proposed. The lower level is implemented as an eBPF program in the Linux kernel, while the upper level is implemented as a user-space agent. TCP retransmit event collection and an event delivery mechanism are implemented.

The practical value of the work lies in the fact that the implemented module makes it possible to detect TCP retransmit events in the Linux kernel, perform their interval-based aggregation, and integrate the results into various analytical systems.

Keywords: eBPF, Linux, network stack, network telemetry, flow telemetry, TCP retransmit, Prometheus, dashboard, JSON snapshots, monitoring.

ЗМІСТ

ВСТУП

Сучасні інформаційні системи залежать від стабільної, передбачуваної та високопродуктивної роботи мережевої підсистеми. Значна частина серверних застосунків, хмарних сервісів, платформ контейнеризації, систем балансування навантаження, мережевих шлюзів, засобів віддаленого доступу та інструментів спостережуваності функціонує саме на платформі Linux. Через це стан мережевого стеку Linux впливає на швидкодію прикладних сервісів, якість обробки запитів, затримку відповіді, стійкість до пікових навантажень і загальну надійність цифрової інфраструктури.

На практиці проблеми мережевої продуктивності рідко проявляються як повна відмова. Значно частіше спостерігається поступова деградація, яка проявляється у вигляді зростання затримки, появи хвостових значень, нестабільної пропускної здатності, сплесків retransmit-подій, збільшення queueing delay та зростання витрат процесорного часу на обробку трафіку. Для своєчасного виявлення проблем потрібна більш детальна телеметрія, прив’язана до конкретних мережевих потоків, процесів, інтерфейсів і часових інтервалів.

Класичні інструменти моніторингу мережі, такі як tcpdump, ss, iproute2, perf і ftrace, добре підходять для разової діагностики, аналізу інцидентів і дослідницьких вимірювань. Проте для безперервного спостереження в робочому середовищі вони не завжди є оптимальними. В цих умовах все більшої актуальності набуває eBPF як механізм безпечного й ефективного розміщення невеликих програм безпосередньо в ядрі Linux. Такий підхід дозволяє збирати потрібні події максимально близько до джерела їх виникнення, зменшувати надлишкове копіювання даних у простір користувача та передавати назовні вже компактний і корисний результат.

Актуальність теми полягає в тому, що сучасні Linux-системи потребують інструментів спостережуваності, які поєднують точність вимірювання, низькі накладні витрати, можливість інтеграції із системами моніторингу та засобами візуалізації, а також підтримують раннє виявлення деградацій. Важливо фіксувати

факт проблеми та одночасно пов’язувати спостережувані симптоми з конкретними подіями мережевого стеку. Тому розроблення програмного модуля моніторингу на базі eBPF для підвищення мережевої продуктивності Linux є актуальним завданням.

Метою кваліфікаційної роботи є розроблення програмного модуля моніторингу на базі eBPF, який забезпечує спостережуваність мережевого стеку Linux для подальшого підвищення мережевої продуктивності.

Для досягнення поставленої мети необхідно виконати такі завдання:

– провести аналіз мережевої телеметрії та flow-телеметрії в Linux;

– проаналізувати інструменти моніторингу мережевого стеку Linux та оцінити доцільність використання eBPF;

– сформувати постановку задачі розроблення модуля моніторингу;

– розробити архітектуру програмного модуля моніторингу мережевого стеку Linux на базі eBPF з урахуванням локального збору телеметрії та можливості віддаленого спостереження через системи моніторингу;

– розробити алгоритми функціонування ключових компонентів модуля, включно з обліком retransmit-подій, інтервальною агрегацією та формуванням зведених метрик;

– розробити інформаційне забезпечення модуля, визначивши структуру подій, формат метрик, правила зберігання та очищення записів потоків;

– провести вибір програмних засобів для реалізації eBPF-частини, користувацького агента, експорту в Prometheus та засобів веб-візуалізації;

– реалізувати ключові модулі системи, а саме eBPF-програму, механізми BPF maps та ring buffer, а також користувацький агент для приймання, агрегації й експорту подій;

– реалізувати інтерфейс взаємодії з користувачем і системами моніторингу через CLI, Prometheus endpoint, JSON snapshots та dashboard;

– провести тестування працездатності модуля та виконати експериментальну оцінку його роботи в типових сценаріях навантаження.

Об’єктом дослідження є процеси збору, обробки, агрегації та інтерпретації телеметрії мережевого стеку Linux у робочому середовищі.

Предметом дослідження є методи і засоби побудови програмного модуля моніторингу на базі eBPF, структури подій ядра Linux, алгоритми інтервальної агрегації телеметрії, а також способи експорту та візуалізації результатів моніторингу.

В роботі використано методи структурно-функціонального аналізу, логіко-алгоритмічного моделювання та порівняльного аналізу засобів моніторингу.

Практичне значення одержаних результатів полягає в тому, що реалізований модуль дозволяє виявляти події TCP retransmit в ядрі Linux, виконувати їх інтервальну агрегацію, формувати короткі CLI-звіти, експортувати метрики у форматі Prometheus, зберігати JSON-знімки стану та відображати результати через веб-панель моніторингу.

Кваліфікаційна робота складається зі вступу, трьох розділів, висновків, списку використаних джерел та додатків.

1 СТАН І АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Мережева телеметрія та телеметрія потоків у Linux

Мережева телеметрія в Linux дає змогу оцінювати фактичний стан передавання даних як на рівні застосунку, так і всередині мережевого стеку. В сучасних системах проблеми продуктивності часто проявляються не як повна відмова, а як повільна деградація. Це може бути коротке зростання затримки, хвилі ретрансляцій, періодичне просідання пропускної здатності або зростання витрат CPU на обробку трафіку. Через це важливо збирати метрики не епізодично, а постійно і прив’язувати їх до конкретних потоків трафіку, інтерфейсів, процесів і моментів часу. Такий підхід відповідає потоковій телеметрії, де потік розглядається як набір пакетів, що належать до одного логічного обміну даними. Найчастіше потік визначають через п’ять ознак: IP джерела, IP призначення, порт джерела, порт призначення, протокол. В Linux це відповідає з’єднанню на рівні сокета для TCP або потоку UDP пакетів для QUIC. Потокова телеметрія дозволяє визначити, в яких потоках формується хвіст розподілу p99, на якому інтерфейсі зростає черга, які з’єднання мають ретрансляції та які процеси створюють найбільше навантаження.

Коли пакет приходить з мережі, він спочатку потрапляє на мережеву карту, далі в драйвер і в підсистему прийому ядра. В [1] описано механізм NAPI, який є ключовим для прийому трафіку під навантаженням. Суть цього механізму полягає в тому, що ядро переходить від переривального режиму до режиму опитування. Це зменшує накладні витрати на обробку переривань, але створює додаткові часові затримки, бо пакет може чекати, поки його обробить цикл опитування і планувальник. Тому планування виконання та взаємодія softirq з планувальником процесора можуть бути одними з джерел затримок, особливо на системах з високим PPS або з конкуренцією за CPU.

Після прийому пакет проходить мережевий стек ядра. Для IPv4 або IPv6 він проходить розбір заголовків, маршрутизацію, фільтрацію і передається далі або в локальний стек, або на пересилання. Якщо пакет адресований локальному хосту і це TCP, то він потрапляє в TCP підсистему, де відбувається обробка стану

з’єднання, підтвердження, контроль перевантаження і розміщення даних в буфери сокета. Потім дані стають доступними для процесу користувача через системні виклики читання. Проблема в тому, що затримка може виникати на кожному етапі. Частина затримки пов’язана з чергами в драйвері або в qdisc, а частина пов’язана з логікою контролю перевантаження, коли відправник не може збільшити швидкість через обмеження cwnd або через оцінку пропускної здатності. Частина пов’язана з ретрансляціями, коли сегмент губиться або приходить із затримкою і TCP змушений повторити передачу. Крім того є зв'язок з scheduling, коли процес отримувача не отримує CPU вчасно і дані довше знаходяться у receive buffer.

Окремо треба виділити найранішу точку доступу до пакета в мережевому шляху. В [2] описано XDP як механізм швидкої обробки пакетів в ядрі. XDP (рисунок 1.1) вбудований в шлях прийому після драйвера, що дає можливість виконувати фільтрацію, підрахунки та вимірювання з мінімальними накладними витратами. Тобто можна точно зафіксувати час входу пакета в систему ще до того, як він пройде ланцюжки обробки. Це допомагає розділити затримку мережі як таку і затримку всередині хоста.

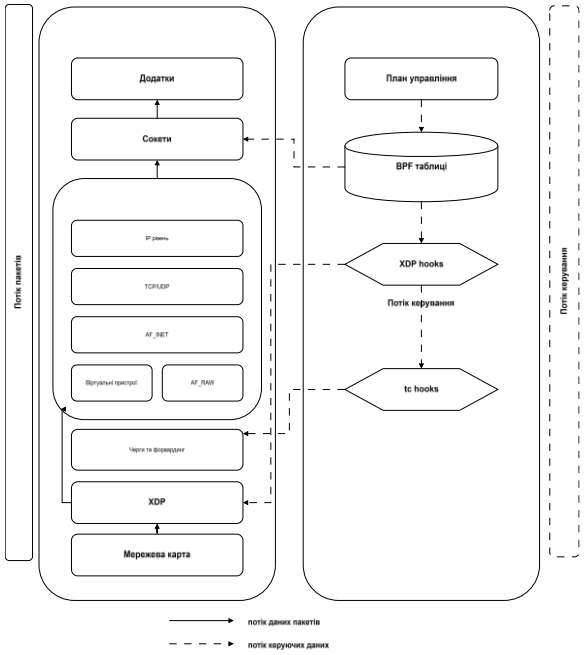


Рисунок 1.1 - Місце XDP на шляху пакета на рівні сокета

Одним з найбільш частих джерел великих затримок є черги. Queueing delay виникає тоді, коли пакети накопичуються в буферах і чекають своєї черги на

обробку або передачу. Ідея полягає в тому, що великі буфери в мережевих пристроях і в ОС можуть приховувати втрати і давати високу пропускну здатність, але за це система платить великим зростанням затримки і джитера. Для телеметрії це означає, що недостатньо оцінювати лише пропускну здатність. Потрібно також вимірювати затримку та аналізувати хвіст її розподілу, бо користувачі часто помічають саме рідкі, але дуже повільні відповіді.

В роботі [3] CoDel описано як алгоритм активного керування чергою. Його робота ґрунтується на контролі часу перебування пакетів у черзі, а не лише на її довжині. Це важливо для Linux, бо qdisc є реальним компонентом в шляху пакета, а не теоретичною частиною. В іншому описі [4] описано реалізацію fq codel у вигляді qdisc. Тут поєднується справедливе розділення потоків і контроль затримки.

Друга велика група причин затримки пов’язана з congestion control і ретрансляціями. В роботі [5] розглядається BBR, як підхід до контролю перевантаження, який орієнтується на оцінку пропускної здатності і затримки. Якщо алгоритм допускає надмірний обсяг непідтверджених даних, це призводить до зростання черг і збільшення RTT. Якщо алгоритм занадто обережний, throughput падає. В роботі [6] показано експериментальні порівняння і роботу BBR в умовах різних буферів.

Ретрансляції є сигналом проблем або в каналі, або в перевантаженні, або в некоректних таймерах. Для TCP це можна спостерігати через події retransmit, через зміну SACK, через оцінки RTO і через лічильники втрат. Для flow телеметрії важливо ідентифікувати такі події до конкретного з’єднання.

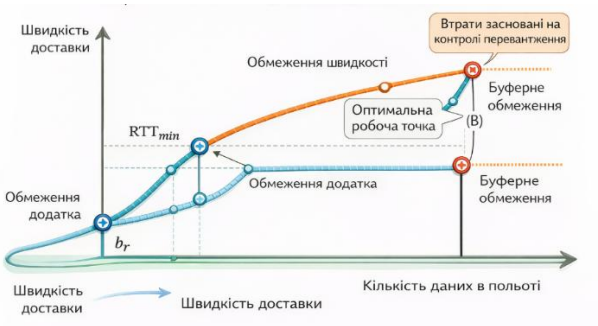


Рисунок 1.2 - Контроль робочих точок при перевантажені та балансування між затримкою і продуктивністю

Під час аналізу продуктивності мережі потрібно наперд визначити, які саме показники будуть використовуватися для оцінювання її стану. Найчастіше враховують затримку, пропускну здатність, втрати пакетів і джитер, а для Linux-систем додатково важливими є витрати процесорного часу. В роботі [7] пояснюється, чому tail latency стає критичною при масштабуванні сервісів і чому p95 та p99 краще відображають якість. Для мережевої телеметрії це означає, що модуль моніторингу повинен будувати розподіли і вміти показати p50 як типову затримку, p95 як затримку для більшості запитів і p99 як показник хвостових затримок, що найчастіше погіршують якість роботи сервісу.

Пропускна здатність є другою основною метрикою, яка показує, який обсяг даних фактично передається за одиницю часу. Проте без врахування супровідних показників пропускна здатність може давати хибне уявлення про стан системи. Висока пропускна здатність може супроводжуватися значною затримкою в чергах. Низька пропускна здатність може бути наслідком механізмів контролю перевантаження, втрат пакетів, малого розміру вікна, перевантаження процесора або того, що прикладна програма не встигає зчитувати дані із сокета. Тому потрібно одночасно враховувати швидкість доставки даних, RTT, втрати та витрати процесорного часу. В роботах [8] та [9] показано, що контроль перевантаження фактично працює у площині двох показників — швидкості та затримки, а зміна буферів або мережевих умов зміщує робочий стан системи в межах цієї площини.

Джитер — це коливання затримки, яке може виникати через черги, нестабільну роботу планувальника, зміну маршруту або погіршення стану каналу. Для Linux потрібно врахувати, що частина джитера може з’являтися через NAPI та softirq, коли під навантаженням пакети обробляються порціями. В [10] представлено, що система спеціально використовує пакетну обробку, і це має побічний ефект у вигляді нерівномірних затримок. Витрати CPU в мережевій підсистемі — це процесорний час, який витрачається на приймання, обробку, копіювання, шифрування, роботу мережевого стеку та контекстні перемикання. Якщо процесор стає вузьким місцем, в системі можуть збільшуватися черги та зростати затримка. Тому потрібно збирати не тільки мережеві лічильники, але і

прив’язку до CPU, наприклад на якому ядрі обробляється трафік, як часто відбуваються softirq, як змінюється завантаження.

1.1.1 TCP і QUIC

QUIC побудований поверх UDP і часто реалізований у просторі користувача, але логіка втрат і контролю перевантаження в ньому є, просто вона інша за деталями. В [11] наведено огляд QUIC в реальних мережах і показано, що вимірювати треба час встановлення з’єднання, кількість обмінів до появи перших корисних даних, оцінки RTT, втрати і повторні передачі на рівні QUIC пакетів, а також події, які відповідають відновленню після втрат (рисунок 1.3).

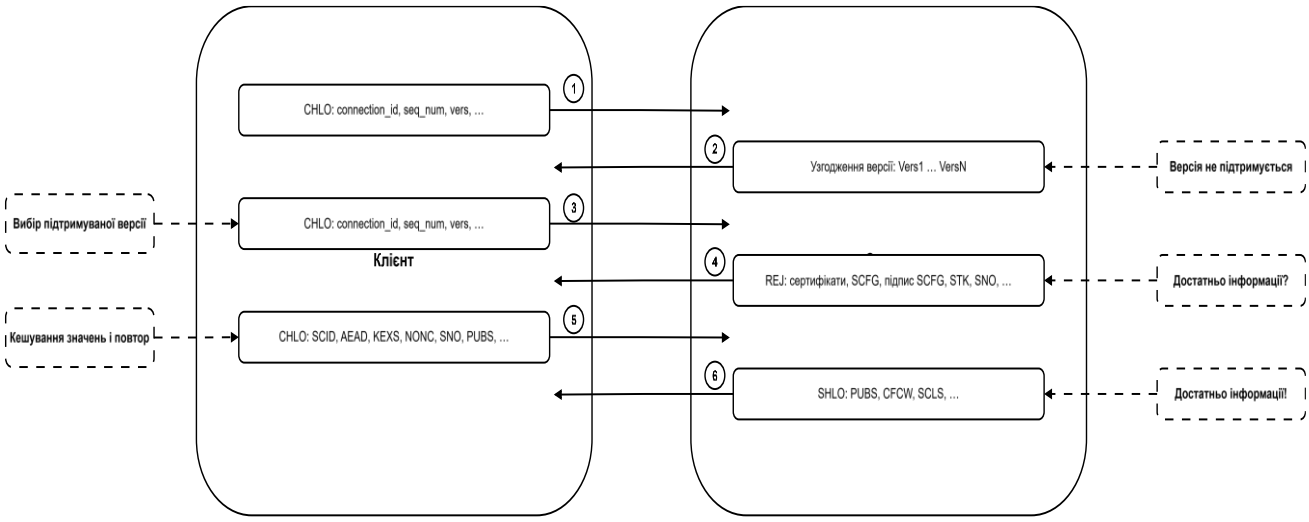


Рисунок 1.3 – Різниця між типовим встановленням зв’язку та з додатковими кроками

Для QUIC важливим джерелом є [12]. В ньому описано механізми виявлення втрат, обробку підтверджень, таймери і стани контролю перевантаження. QUIC має власні поняття втрати і повторної передачі, які не зводяться до TCP retransmit. В QUIC повторна передача не завжди означає повторення того самого пакету, в більшості випадків це повторне надсилання даних в іншому пакеті. Тому модуль моніторингу має збирати метрики на рівні UDP-потоку та, за потреби, доповнювати їх даними з простору користувача. Також в RFC 9002 описана схема станів, яка пояснює як алгоритм переходить між цими станами.

1.2 Інструменти моніторингу мережевого стеку Linux

Адміністрування і розробка мережевих сервісів в Linux розділені на два підходи до спостережуваності. Перший підхід моніторить зовнішній трафік через захоплення пакетів та аналіз заголовків. Другий підхід фокусується на мережевому стеку зсередини через стан сокетів, події ядра і профілювання.

Захоплення пакетів дає найбільш буквальну картину того, що пройшло по інтерфейсу, а трасування ядра дозволяє зрозуміти, чому саме виникла затримка, ретрансляція або падіння пропускної здатності.

Одним із найвідоміших класичних інструментів мережевого аналізу є tcpdump. Він працює як консольний аналізатор пакетів і зчитує трафік через libpcap [13], де підкреслюється, що tcpdump використовує libpcap як незалежний від платформи інтерфейс захоплення пакетів. Ідея libpcap як переносимого шару, полягає в тому, що він приховує різні механізми захоплення пакетів, оскільки різні системи мають різні інтерфейси і можливості. Тобто tcpdump і libpcap формують класичний підхід до захоплення пакетів, де основний обсяг роботи з декодуванням і фільтрацією відбувається на стороні користувацького простору, а ядро надає трафік через capture інтерфейс.

Важлива деталь класичного підходу це фільтрація трафіку в ядрі і для цього використовувався Berkeley Packet Filter. В [14] описано архітектуру BPF як механізм, що дозволяє виконувати байткод фільтра в ядрі та відсікати непотрібні пакети ще до копіювання у користувацький простір. Саме ця ідея розвинулась в eBPF.

Якщо знімати весь трафік без фільтра, зростає обсяг копіювання та обробки і це може змінити поведінку системи. Тому точність tcpdump з точки зору змісту пакетів висока, але накладні витрати на використання ресурсів, особливо на лініях 10Gbps будуть високими. В роботі [15] tcpdump розглядався, як базовий інструмент мережевого аналізу і там же акцентується, що він добре підходить для пошуку помилок і аналізу пакетів, але за своїм призначенням він більше підходить для діагностики, ніж для постійного збору телеметрії в робочому середовищі.

До другої класичної групи інструментів відносяться утиліти iproute2, зокрема ss. Вони не захоплюють пакети напряму, а читають стан ядра через системні інтерфейси і показують його у зрозумілому вигляді. В роботі [16] наведено режим показу внутрішньої TCP інформації де вказуються параметри таймерів, оцінки rtt, rttvar, значення rto, а також назву алгоритму контролю перевантаження. Це означає, що ss може показати багато важливих TCP метрик без дампу трафіку. Такий підхід зазвичай дешевший за tcpdump, бо він не копіює кожен пакет. Проте він має інше обмеження. Він дає зріз стану, а не точний часовий ряд подій. Якщо потрібно зафіксувати короткі піки затримки, раптові сплески повторних передач або проблему, яка проявляється лише періодично, одного зрізу стану недостатньо. В роботі [17] цей клас інструментів протиставляється packet capture підходу, бо вони зручніші для швидкої перевірки стану сокетів і черг, але не замінюють глибокого аналізу, коли потрібна деталізація по пакетах або функціях ядра.

Профілювання і трасування ядра в класичному вигляді реалізується через perf та ftrace. У [18] perf описується як інструмент, який вміє працювати з апаратними лічильниками та точками трасування, динамічними точками в ядрі та точками спостереження в користувацькому просторі, і може виконувати профілювання з невеликими накладними витратами. Це важливо для мережевих задач, тому що в Linux існують точки трасування для TCP і мережевого шляху, а динамічні точки в ядрі дають змогу підключатися до потрібних функцій ядра. В роботі [19] наведено карту основних засобів трасування, де згадуються ftrace, точки трасування, динамічні точки в ядрі, кільцевий буфер, а також підходи до аналізу поведінки через події. В сукупності це показує, що perf і ftrace дають доступ до внутрішніх подій ядра, які недоступні tcpdump, наприклад до подій обробки сокетів, таймерів TCP або місць, де затримка виникає через планувальник.

Окреме питання для perf і ftrace це накладні витрати. В [20] розглядається тема точності та overhead підсистеми perf events, і наведено оцінки часу виконання операцій Start, Stop та Read для лічильників. Це підтверджує, що глибоке трасування може бути витратним (рисунок1.4), якщо збирати надто багато подій,

зберігати стек викликів для кожної події або проводити відбір зразків із надто малою паузою.

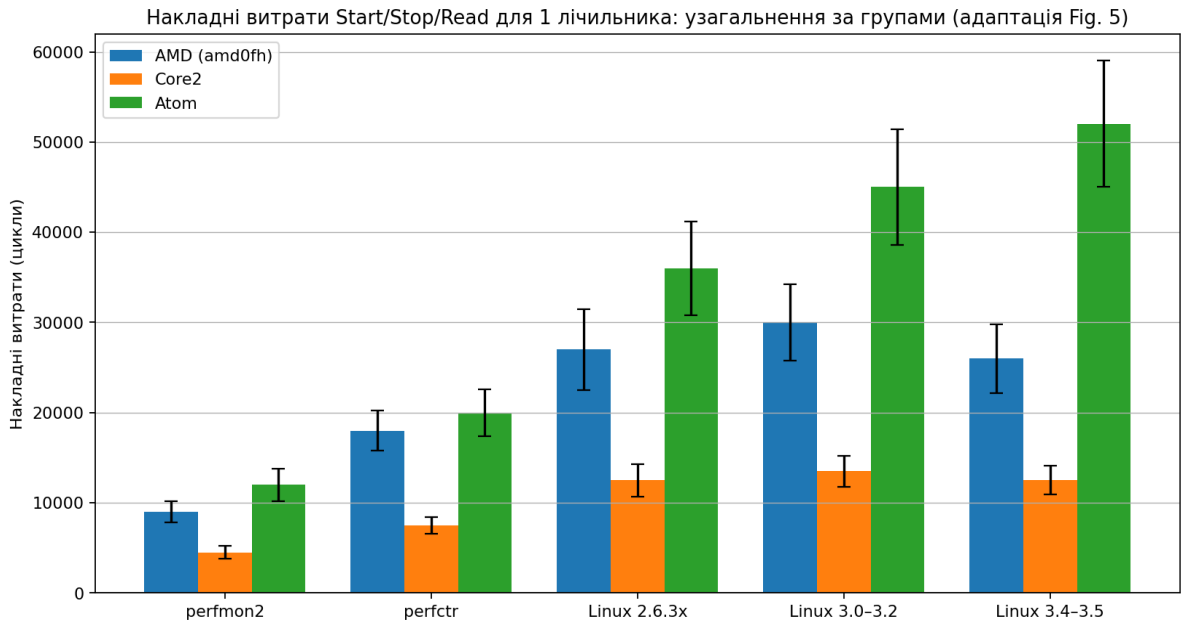


Рисунок 1.4 – Узагальнені накладні витрати perf events під час операції Start Stop Read для одного лічильника продуктивності на різних платформах

Тобто класичні інструменти трасування добре підходять для налагодження та аналізу, але для постійного моніторингу в робочому середовищі потрібні обмеження, фільтри і раціональна частота збору, інакше спостережуваність сама стане фактором деградації.

Екосистема eBPF пропонує компроміс між глибиною і вартістю спостереження. В [21] eBPF описується як інструмент, який дозволяє вбудовувати байткод в ядро і збирати метрики з низькими накладними витратами. Там же порівнюються інструменти BCC і bpftrace як два стилі роботи.

BCC орієнтований на готові утиліти і більш структуровані програми, а bpftrace орієнтований на швидку разову інструментацію. Крім того в даній роботі описано варіант, де модулі вбудовують eBPF (рисунок 1.5) програми, далі метрики збираються і передаються в шар візуалізації. Це показує, що eBPF можна не просто запустити як разовий скрипт, а вбудувати у стандартний стек моніторингу.

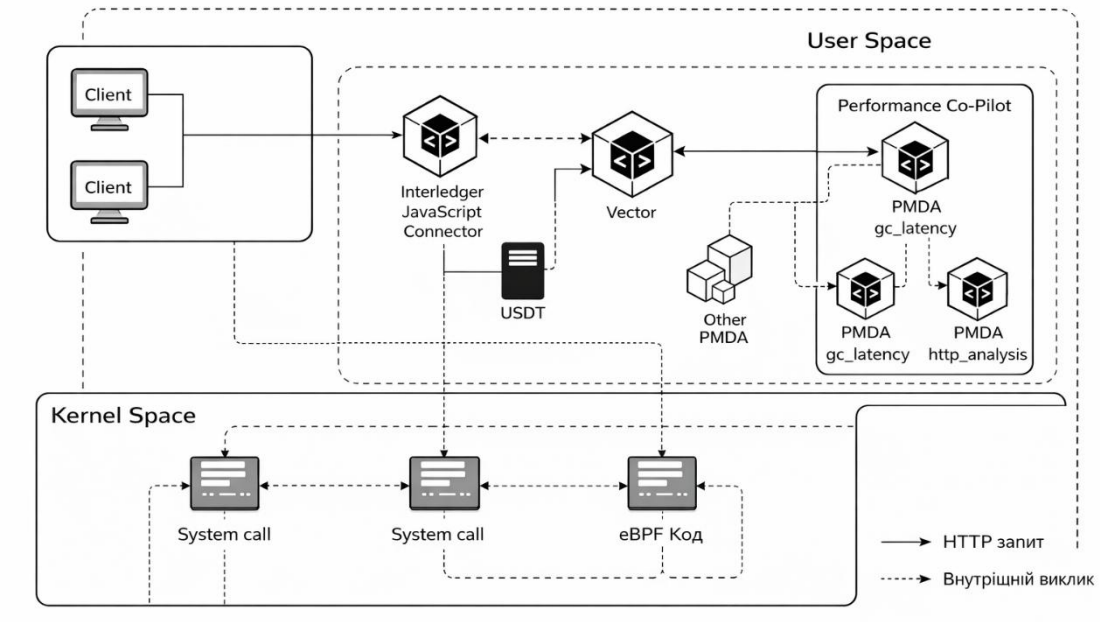


Рисунок 1.5 – Узагальнена схема налаштування моніторингу продуктивності на базі eBPF в Linux

В [22] наведено перелік готових інструментів, серед яких є утиліти для мережі такі як tcpretrans для трасування ретрансляцій TCP. Це важливо, бо саме готові утиліти часто стають початковою основою для побудови рішень у робочому середовищі. Спочатку береться існуючий інструмент з BCC, перевіряються метрики і точки підключення, після чого логіка переноситься у власний агент, який експортує дані у потрібний формат. Для робочого середовища це вигідно тим, що логіка збору працює в ядрі, а в user space можна залишити лише агрегацію, зберігання та відправку.

В роботі [23] представлено приклад реалізації мережевої телеметрії на базі eBPF. В ній реалізовано ePPing (рисунок 1.6), який вимірює RTT пасивно, без активних ICMP запитів, а вся логіка парсингу пакетів, зіставлення відповіді з запитом і обчислення RTT працює в ядрі, а користувацький компонент лише завантажує програму і отримує результати. Також автори роботи зазначають, що версія на базі eBPF здатна обробляти високошвидкісний трафік з меншими накладними витратами. З цього випливає, що eBPF часто виграє у tcpdump для постійної телеметрії. Tcpdump дає кожен пакет і перекладає весь аналіз на

користувацький простір, а eBPF дозволяє робити частину аналізу до копіювання і одразу формувати метрику, наприклад RTT.

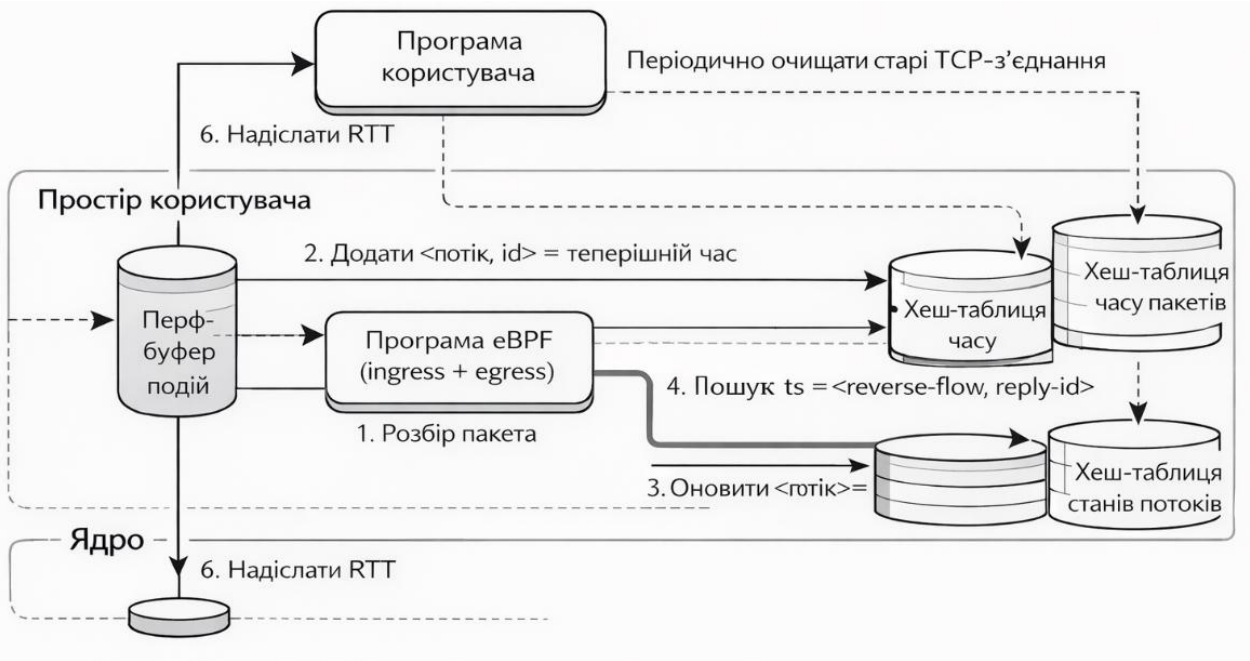


Рисунок 1.6 – Схема ePPing для пасивного вимірювання RTT на базі eBPF

Ще одним напрямом сучасних рішень є використання телеметрії, як основи для оптимізації мережевих алгоритмів. В роботі [24] розглянуто ідею керування перевантаженням на основі телеметрії, де eBPF використовується для збирання сигналів з боку хоста і допомагає робити більш інформований контроль перевантаження. Це показує можливість використання моніторингу і для прийняття рішень про налаштування параметрів ядра. В іншій роботі [25] розглянуто інший клас задач, пов'язаний із внутрішнім аналізом трафіку в ядрі, де eBPF використовується для побудови компактних структур даних, які оцінюють характеристики потоків і дають гарантії точності при малому використанні пам'яті. Це показує, що телеметрія, придатна для використання в робочому середовищі може бути агрегованою, але при цьому корисною для виявлення важких потоків, аномалій і змін в навантаженні.

Потрібно підкреслити, що eBPF не є універсальним рішенням без накладних витрат. Якщо eBPF програма виконується на кожному пакеті або на кожному системному виклику, то навіть невелика робота множиться на велику частоту

подій. Тому в сучасних роботах багато уваги приділяється контролю overhead. В роботі [26] досліджено проблему накладних витрат для процесів, які не є безпосередньою ціллю трасування, і пропонується підхід, який ізолює витрати до конкретного процесу. Також в цій роботі показано, як знімати метрики для latency sensitive сценаріїв і як вибирати точки інструментації так, щоб не створити надмірний вплив на систему. Ще в одному дослідженні [27] розглядається низьковитратне трасування на рівні сокетів. Чим ближче інструмент до критичного шляху, тим жорсткіші вимоги до фільтрації, семплювання та агрегації.

Інтеграція eBPF з реальними протоколами і системами теж має значення для робочих середовищ. В роботі [28] показано, як eBPF інтегрується в протоколи з підтримкою модулів розширення протоколів, зокрема в контексті QUIC, через anchor points і керовані виклики. Це не є утилітою на зразок tcpdump, але демонструє, що eBPF використовується як універсальний механізм безпечного розширення та спостереження. В роботі [29] описано trace driven emulation для супутникових мереж з використанням eBPF, тобто eBPF застосовують і для збору трас, і для відтворення умов. Такі роботи підтверджують, що eBPF став базовою платформою для нових інструментів спостережуваності, а не лише модою для окремих утиліт.

Практична придатність до використання в робочому середовищі часто пов’язана не тільки з продуктивністю, а й з безпекою та простотою експлуатації. В роботі [30] описано безпековий моніторинг інформаційних систем за допомогою eBPF. В [31] описано застосунок моніторингу трафіку та визначення DDoS атак з допомогою eBPF, тобто коли eBPF використовується як постійний датчик. В іншій роботі [32] детально розглядаються методології використання eBPF для моніторингу і виявлення шкідливої активності, включно з практичними модулями і кодами. Вони підтверджують можливість використання eBPF як механізму, що може працювати постійно і збирати події з ядра без модифікації ядра.

Якщо порівняти рішення за критеріями точність, накладні витрати, інтеграція та реальне застосування, то картина виглядає так: tcpdump і libpcap дають максимальну буквальність на рівні пакета і дозволяють перевірити протокол,

прапори, повтори, порядок сегментів. Проте вони легко стають дорогими на великих швидкостях і вимагають збереження або обробки великого обсягу даних.

ss і iproute2 дають дешевший доступ до стану TCP, але це скоріше знімок параметрів, а не повна історія.

perf і ftrace дають глибину і дозволяють пояснити поведінку ядра, але вимагають досвіду, контролю overhead, що підтверджується оцінками в [33] та описом можливостей в [34]. eBPF дозволяє перемістити частину аналізу ближче до джерела подій, тобто в ядро, і тим самим зменшити копіювання і підвищити якість метрик. Водночас eBPF потребує більш обережного підходу до фільтрації подій та контролю накладних витрат. Для інтеграції з observability перевагою eBPF є можливість будувати стабільний конвеєр збору і експорту метрик в Prometheus та візуалізації для Grafana.

Аналіз розглянутих робіт показує, що класичні інструменти залишаються незамінними для точного розбору пакетів і швидкого перегляду стану сокетів, але вони не завжди підходять для постійної телеметрії і оптимізації в режимі онлайн. eBPF рішення, зокрема BCC та bpftrace, дають більш гнучкий і масштабований шлях до метрик затримок, ретрансляцій і продуктивності ядра, а також краще інтегруються в сучасні конвеєри спостережуваності. Саме тому за основу краще брати eBPF, а класичні інструменти використовувати як контрольні та допоміжні під час валідації і дослідженні інцидентів.

1.3 Постановка задачі

Проведений аналіз існуючих рішень показав, що для мережевого стеку Linux типові проблеми продуктивності найчастіше проявляються як поява дуже великих затримок, сплески ретрансляцій, нестабільна пропускна здатність та збільшення витрат CPU на обробку трафіку. Класичні інструменти, такі як tcpdump, ss, iproute2, perf і ftrace, добре підходять для ручної діагностики та разових досліджень. Але для постійного контролю в реальних системах їх потрібно використовувати обережно через накладні витрати та складність інтеграції.

В розділі 1.2 показано, що глибоке трасування може бути дорогим, якщо збирати забагато подій або робити надто частий відбір зразків, тому для постійного моніторингу потрібні обмеження та раціональна частота збору. Одночасно екосистема eBPF дає можливість змістити частину обробки ближче до ядра і збирати метрики з меншими накладними витратами, а також вбудовувати збір у стандартний конвеєр спостережуваності.

Метою роботи є розроблення програмного модуля моніторингу на базі eBPF, який забезпечує спостережуваність мережевого стеку Linux для подальшого підвищення мережевої продуктивності за рахунок керованого налаштування параметрів системи. Модуль має виконувати безперервний збір телеметрії з мінімальним впливом на роботу хоста, надавати метрики в зручному форматі та підтримувати сценарії діагностики і контролю деградацій.

Вимоги до модуля визначаються тим, які події він спостерігає і які метрики здатен обчислювати. Базовим є збір подій, що відображають роботу TCP на рівні ядра: встановлення з’єднання, завершення з’єднання, зміну стану, ретрансляції та ознаки втрат. Для оцінки затримок модуль має формувати часові мітки на подіях відправлення та отримання і корелювати їх на рівні потоку, [23]. Для QUIC, який часто реалізований у просторі користувача поверх UDP, модуль має підтримувати flow телеметрію для UDP, а для точнішої інтерпретації подій передбачити інтеграцію з користувацькими маркерами або USDT, якщо вони доступні в застосунку, оскільки практичні конвеєри спостережуваності можуть поєднувати користувацькі події та eBPF програми [21] .

На основі зібраних подій модуль має рахувати метрики продуктивності мережі у вигляді часових рядів і агрегатів. Метрики мають бути доступні як для всієї системи, так і з привязкою до потоку або до групи потоків за фільтром, наприклад за портом, інтерфейсом або процесом.

Модуль не повинен помітно впливати на затримку та пропускну здатність робочих сервісів. Для цього потрібні фільтрація та агрегація на рівні ядра, обмеження частоти подій, уникнення важких операцій на кожному пакеті та можливість ввімкнути режим відбору зразків.

Модуль має допомагати швидко визначати, чи пов’язана проблема: з чергами, ретрансляціями, механізмами контролю перевантаження або нестачею процесорних ресурсів для обробки трафіку.

На основі зібраних метрик модуль має дати можливість перевіряти ефект змін, розмірів буферів сокетів, вибору алгоритму контролю перевантаження, параметрів черг, і порівнювати стан до і після.

Модуль має працювати як ранній індикатор, який показує тренд погіршення, ріст ретрансляцій, ріст витрат CPU, і виявляти проблему до того, як вона стане помітною користувачам.

Для досягнення поставленої мети необхідно виконати наступні завдання:

1. Розробити архітектуру програмного модуля моніторингу мережевого стека Linux на базі eBPF з врахуванням локального збору телеметрії та можливості віддаленого спостереження через системи моніторингу.

2. Розробити алгоритми функціонування ключових компонентів модуля, включно з кореляцією подій для оцінки затримок, обліком ретрансляцій та агрегацією метрик.

3. Розробити інформаційне забезпечення модуля, визначивши структуру подій, формат метрик, правила зберігання та очищення записів потоків.

4. Провести вибір програмних засобів для реалізації eBPF-частини та користувацького агента, а також вибір інструментів інтеграції з Prometheus і засобів візуалізації.

5. Реалізувати ключові модулі, зокрема eBPF програми для збору подій, механізми BPF maps та буферів подій, а також користувацький компонент для агрегації і експорту.

6. Реалізувати інтерфейси взаємодії з користувачем і системами моніторингу через Prometheus endpoint, JSON-вивід та CLI-звіти.

7. Протестувати працездатність модуля та провести експериментальну оцінку точності, накладних витрат, стабільності і відтворюваності результатів у типових сценаріях навантаження.

2 АЛГОРИТМІЧНЕ ТА ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ МОДУЛЯ

2.1 Загальна структура проектованого модуля

Розроблюваний модуль призначений для постійного збору мережевої телеметрії в Linux без модифікації ядра та без потреби знімати повні дампи пакетів. У попередніх розділах було показано, що класичні інструменти дають багато корисної інформації, але в реальних застосуваннях їх важко тримати увімкненими постійно через вартість збору, обсяг даних і складність інтеграції. При цьому сучасні підходи на базі eBPF дозволяють перенести частину обробки ближче до ядра, виконувати ранню фільтрацію і формувати метрики одразу під час події, а вже в користувацькому просторі залишити агрегацію і експорт. Такий підхід відповідає ідеї моніторингу з мінімальним впливом на систему, де інструмент має мінімально впливати на продуктивність і працювати як частина стандартного стека спостережуваності [21]. Додатково архітектура має врахувати, що будь який механізм трасування має накладні витрати, і ці витрати потрібно контролювати за рахунок фільтрів, режимів збору та обмеження частоти подій,

Загальна архітектура модуля побудована як конвеєр куди входить нижній рівень, який працює в ядрі, а верхній — у просторі користувача. В ядрі розміщуються eBPF програми, які прикріплюються до вибраних контрольних точок. Для передачі подій в користувацький простір використовується ring buffer або perf buffer, щоб не копіювати великі обсяги і не блокувати критичний шлях.

В користувацькому просторі працює агент, який завантажує eBPF частину, налаштовує фільтри, читає події з буфера, агрегує їх у метрики та експортує дані у зовнішні системи.

В типовому випадку агент відкриває кінцеву точку метрик для Prometheus, може віддавати JSON для локального збору або конвеєрів обробки журналів і має CLI режим для швидкої діагностики. Далі метрики зберігаються у сховищі часових рядів, візуалізуються і можуть запускати попередження при деградаціях як показано на рисунку 2.1.

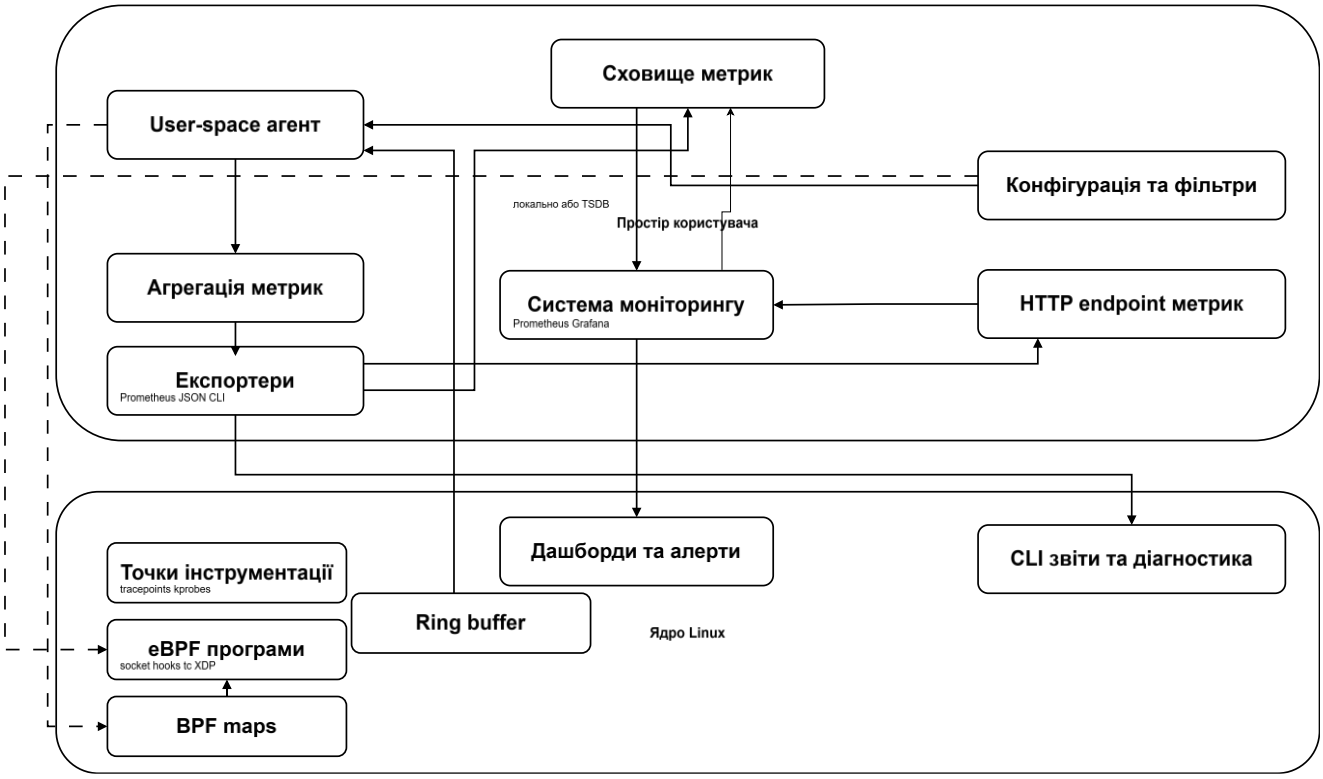


Рисунок 2.1 – Архітектура модуля eBPF

Важливим архітектурним рішенням є розподіл відповідальностей між ядром і користувацьким простором. В ядрі можна залишати лише те, що потребує точного доступу до події і має бути виконано максимально швидко. Це парсинг мінімального набору полів з пакета або з контексту сокета, фіксація часових міток, інкремент лічильників, оновлення стану потоку і відбір подій за фільтром. Зберігання великих структур, складні обчислення перцентилів і формування текстових форматів не доцільно виконувати в ядрі, тому що це збільшує накладні витрати та підвищує ризик переповнення буферів. Тому агрегація, обчислення p50 p95 p99, формування часових рядів, зведення по інтерфейсах і процесах, а також експорт у Prometheus або JSON виконуються в агенті. Такий розподіл дає змогу отримувати точні сигнали з ядра, а обробку, масштабування та гнучке подання результатів виконувати у просторі користувача.

Точки спостереження визначають, які події модуль може бачити і з якою точністю. Для TCP метрик основним джерелом подій є tracepoints, тому що вони є відносно стабільними, мають зрозумілу семантику і добре інтегровані з трасувальними фреймворками ядра (рисунок 2.2).

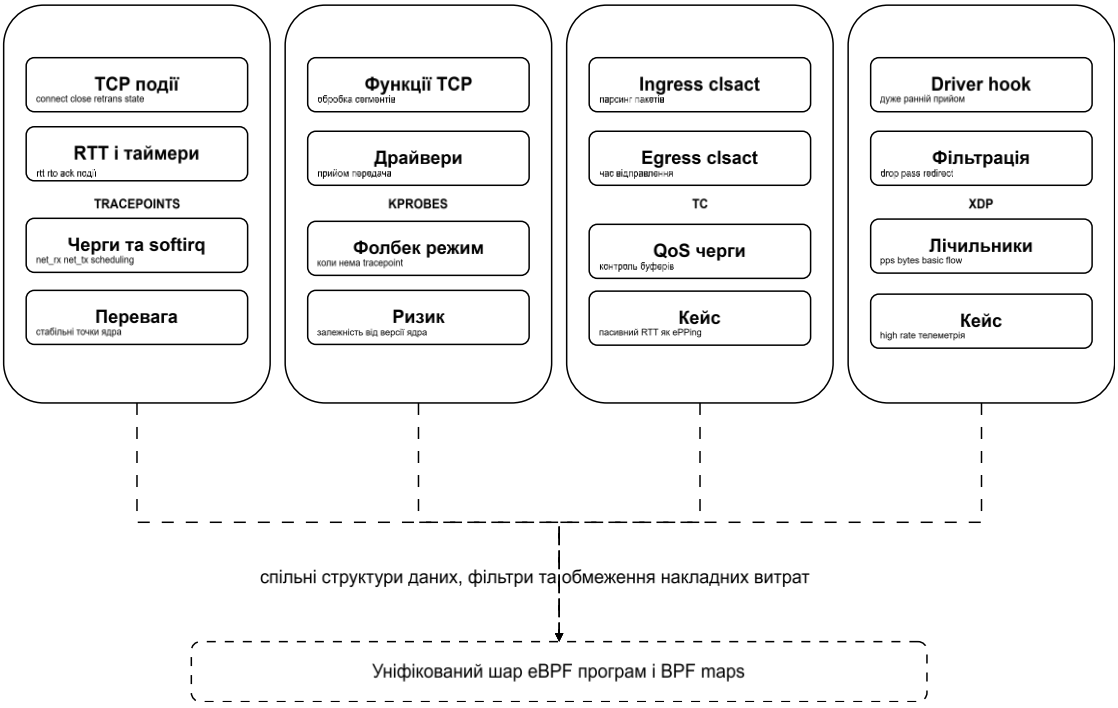


Рисунок 2.2-Точки інструментації tracepoints kprobes tc XDP

Через tracepoints можна відстежувати події встановлення і завершення з’єднання, ретрансляції, зміни стану та інші переходи, не вдаючись до захоплення кожного пакета. Там, де tracepoints не дають потрібної деталізації або відсутні на конкретній версії ядра, як резервний варіант використовуються kprobes.

Окремий клас засобів трасування пов’язаний з мережевими хуками tc та XDP. Їх використовують тоді, коли потрібна пакетна телеметрія на високих швидкостях або коли треба зафіксувати часову мітку максимально близько до входу пакета в стек.

Для модуля XDP розглядається як опція для high rate вимірювань, таких як лічильники пакетів і байтів, базова flow статистика або рання фільтрація. tc краще підходить там, де потрібні ingress і egress точки на рівні інтерфейсу, наприклад для пасивного вимірювання RTT, де потрібно корелювати запит і відповідь у часі. eBPF-програма працює на етапах приймання та відправлення пакетів, зберігає часові мітки у хеш мапах і обчислює RTT, а результат передається у простір користувача через буфер подій.

Тому модуль на рівні ядра може надійно збирати лише базову flow телеметрію UDP, статистику по інтерфейсах та ознаки перевантаження на рівні хоста. Якщо потрібні протокольні метрики QUIC, наприклад точні події loss

detection або стадії handshake, архітектура повинна допускати додатковий шар в агенті, який збагачує мережеві метрики користувацькими маркерами або логами протоколу.

Потоки даних у модулі описуються через два режими: збір окремих подій та інтервальну агрегацію (рисунок 2.3).

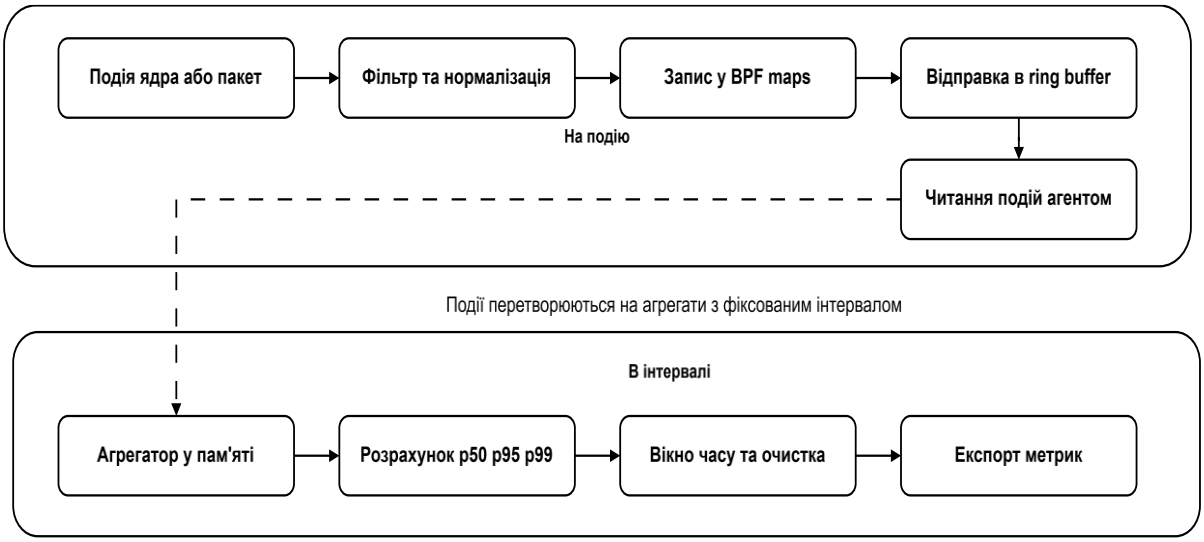


Рисунок 2.3 – Потоки даних на подію та в інтервалі

На подію збираються короткі записи мінімального розміру. Це може бути факт ретрансляції, зміна стану з’єднання, виміряна різниця часу для конкретного ключа потоку, або приріст лічильника байтів.

Далі ці події йдуть в ring buffer, або частина з них залишається тільки у BPF maps у вигляді агрегованого стану.

В інтервалі агент збирає події, оновлює в пам’яті агрегатори, рахує percentiles, формує часові ряди і очищує застарілі записи. Така очистка потрібна для стабільності, тому що мережеві потоки мають великий простір ключів і без контрольованого життєвого циклу таблиці можуть неконтрольовано зростати, що в підсумку призведе до нестабільної роботи модуля.

Архітектура також повинна передбачати механізми контролю накладних витрат (рисунок 2.4). Тобто потрібні фільтри, щоб не виконувати складні дії для всього трафіку, якщо це не потрібно.

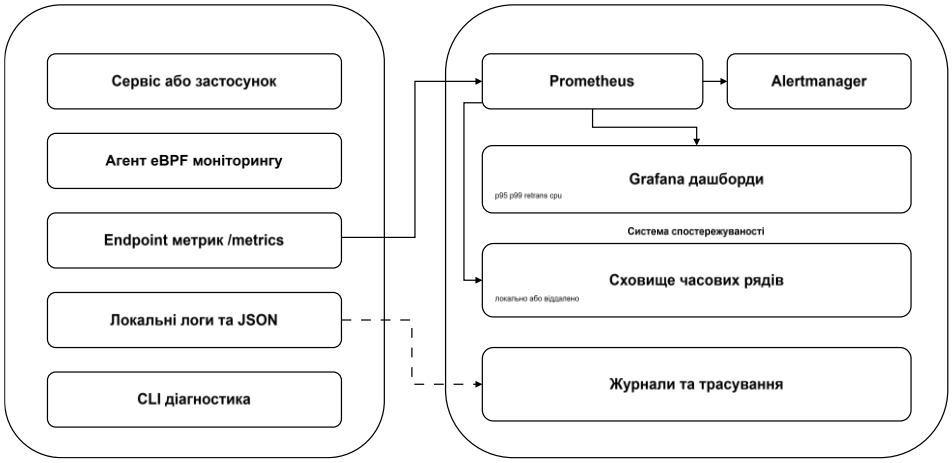


Рисунок 2.4 –Розгортання і інтеграція з observability

Тому агент повинен мати конфігурацію режимів роботи, а також журналювання власного стану, щоб можна було помітити, коли модуль починає втрачати події або виходить за допустимі межі витрат.

Інтеграційний шар у користувацькому просторі будується так, щоб модуль легко вбудовувався в існуючу інфраструктуру. У типовому експлуатаційному середовищі збір метрик робить Prometheus, який періодично опитує endpoint, а візуалізація і попередження формуються у Grafana та суміжних компонентах.

Для локальної роботи, наприклад на стенді або під час інциденту, потрібен CLI-режим, який формує короткий звіт і дозволяє побачити p95 p99, ретрансляції, навантаження CPU та активні проблемні потоки без зовнішніх залежностей.

2.2 Алгоритмічне забезпечення модуля

Основою алгоритмічного забезпечення є алгоритми збору та агрегації (рисунок 2.5). Для TCP модуль повинен підтримувати три групи вимірювань. Перша група це затримка встановлення з’єднання. Друга група це поточна затримка передачі даних, яку зручно оцінювати через RTT або через пов’язані ACK події. Третя група це затримка, пов’язана з повторною передачею, коли модуль фіксує факт retransmit і відстежує, коли проблема зникає або коли пакетний обмін повертається в нормальний стан.

Таке розділення відповідає реальним симптомам деградації. Наприклад, connect latency показує проблеми на початку взаємодії, RTT показує загальну чутливість потоку до черг і перевантаження, а retransmit latency відображає втрати або нестабільність каналу.

Алгоритм збору TCP-затримки починається з вибору подій, які відображають ключові етапи роботи з'єднання.

```
Початок

Для кожної події TCP у ядрі
    Сформувати ключ потоку
    Ключ = адреса джерела + адреса призначення + порт джерела + порт призначення + протокол

    Якщо подія = початок connect
        Зберегти часову мітку у BPF maps

    Якщо подія = встановлення з'єднання
        Якщо для ключа є початкова мітка
            Обчислити connect_latency
            Передати значення у ring buffer
            Видалити початкову мітку

    Якщо подія = вихідний сегмент або контрольна точка для RTT
        Оновити часову мітку потоку

    Якщо подія = ACK або завершення циклу обміну
        Якщо для ключа є попередня мітка
            Обчислити tcp_latency
            Передати значення у ring buffer

    Якщо подія = retransmit
        Збільшити лічильник retransmits для потоку
        Зберегти часову мітку проблемного епізоду

Кінець для

Кожного інтервалу агрегації
    Зчитати значення з ring buffer
    Оновити агрегатори p50 p95 p99
    Оновити лічильники retransmits
    Сформувати метрики для експорту

Кінець
```

Рисунок 2.5 - Псевдокод алгоритму збору та агрегації TCP latency

Для connect latency береться момент початку встановлення зєднання і момент переходу сокета в стан established. В найпростішому варіанті модуль фіксує початкову часову мітку при події connect і зберігає її у BPF maps за ключем потоку. Коли надходить подія успішного встановлення з'єднання, модуль виконує пошук за ключем, обчислює різницю між часовими мітками та формує готове вимірювання. Далі це вимірювання надходить у ring buffer і потрапляє в агент.

Для RTT або ACK орієнтованої оцінки принцип такий самий, але ключове місце займає кореляція подій відправлення і підтвердження. Для цього

використовується мінімальний набір полів потоку - це адреси, порти, напрям і службові ознаки пакета.

Якщо модуль працює через tc ingress та egress або через події TCP, він зберігає часову мітку для вихідної дії і оновлює її при отриманні підтвердження. Для retransmit latency модуль фіксує подію повторної передачі, збільшує лічильник проблем для цього потоку і додатково ставить часову мітку для епізоду.

Коли потік знову проходить через успішну ACK-подію або коли нова телеметрія показує повернення до нормального обміну, агент може оцінити тривалість проблемного інтервалу.

При цьому недоцільно передавати у простір користувача всі події, які фіксує ядро. Якщо модуль пересилатиме кожну проміжну подію, він сам створюватиме додаткове навантаження на систему. Тому алгоритм збору повинен будуватися за принципом мінімальної достатності.

В BPF maps залишаються проміжні часові мітки, лічильники та короткий стан потоку. У ring buffer передаються або завершені вимірювання, або короткі сигнали про важливу подію. Це дозволяє поєднати точність і низькі накладні витрати.

Для QUIC (рисунок 2.6) алгоритм ускладнюється тим, що значна частина логіки працює в просторі користувача поверх UDP.

```
Початок

Для кожної UDP події
    Сформувати ключ потоку
    Ключ = адреса джерела + адреса призначення + порт джерела + порт призначення + протокол UDP

    Якщо подія = передача пакета
        Збільшити лічильник пакетів і байтів
        Зберегти часову мітку останньої передачі

    Якщо подія = прийом пакета
        Збільшити лічильник пакетів і байтів
        Якщо є попередня мітка для цього ключа
            Обчислити інтервал між передачею і прийомом
            Передати значення в агент

    Якщо доступний user space marker
        Прив'язати marker до ключа потоку
        Використати marker для точнішої кореляції handshake або запиту

Кожного інтервалу
    Звести обсяг трафіку по потоку
    Оцінити затримку за часовими інтервалами
    Виявити нестабільні потоки
    Підготувати метрики для експорту

Кінець
```

Рисунок 2.6 - Псевдокод алгоритму збору QUIC телеметрії на рівні UDP

Це означає, що модуль не може покладатися лише на події TCP рівня. Базовий варіант алгоритму повинен вміти спостерігати UDP потоки, групувати їх за набором із п’яти мережевих ознак, рахувати обсяги трафіку, інтервали між передаванням і прийомом, а також формувати ознаки нестабільності.

Якщо застосунок не надає додаткових маркерів, тоді кореляція виконується саме за цим ключем і за часовим вікном. Якщо ж застосунок може віддавати user space marker, наприклад ідентифікатор запиту або маркер стадії handshake, тоді агент поєднує системну телеметрію з логікою протоколу і отримує точніше вимірювання. В базовому режимі це flow телеметрія UDP плюс часове зіставлення. У розширеному режимі це flow телеметрія плюс контекст застосунку.

Після збору подій потрібна агрегація. Вона виконується агентом у фіксованих інтервалах часу. Кожен інтервал є часовим вікном, в межах якого нові події додаються до агрегаторів. Для latency це накопичення вибірки значень, з якої далі оцінюються p50, p95 та p99. Для retransmits це простіше, оскільки достатньо рахувати кількість подій і нормувати її на число активних потоків або на обсяг трафіку. Для throughput береться сумарний обсяг байтів у вікні. Для CPU береться або системна метрика, або окремий показник навантаження softirq та мережевої обробки. Після завершення інтервалу агент формує зведений запис, віддає його у Prometheus і паралельно очищує застарілий стан потоків. Така очистка є обов’язковою, тому що без неї BPF maps і внутрішні таблиці агента будуть постійно зростати.

Окремим алгоритмом є оцінка стану. На цьому рівні модуль оцінює поточний стан мережевої підсистеми. Найпростіший і водночас практичний варіант базується на порогах і евристиках. Агент формує базову лінію для нормального стану і порівнює з нею поточні вікна. Якщо p95 або p99 різко зростає відносно кількох попередніх інтервалів, це вважається сигналом деградації. Якщо одночасно зростає кількість ретрансляцій, проблема найімовірніше пов’язана з втратами пакетів або нестабільністю каналу. Якщо затримка росте, але retransmits залишаються низькими, а throughput не падає різко, це схоже на проблему queueing delay. Якщо зростає завантаження CPU і видно затримки в обробці подій, тоді

можлива деградація на стороні хоста. Такий підхід є відносно простим, але практично корисним, оскільки спирається на зрозумілі симптоми деградації мережі. Для оцінки queueing delay додатково корисно враховувати параметри qdisc. У випадку FQ CoDel важливими є target, interval, memory limit та увімкнення ECN. Це дає можливість повязати проблему з поведінкою черги (рисунок 2.7).

```
Початок

Для кожного нового інтервалу
    Отримати поточні метрики
    Поточні метрики = p50 + p95 + p99 + retransmits + throughput + cpu

    Отримати базову лінію з попередніх інтервалів

    Якщо p95 або p99 зросли більше за допустимий поріг
        Позначити стан як підозрілий

    Якщо retransmits зростають у кількох інтервалах підряд
        Позначити можливу втрату або нестабільність каналу

    Якщо latency зростає, а retransmits не ростуть
        Позначити можливий queueing delay

    Якщо cpu зростає і одночасно росте latency
        Позначити можливе перевантаження хоста

    Якщо одночасно спрацювало кілька умов
        Позначити стан як деградацію
        Передати сигнал у модуль рекомендацій

Кінець для

Кінець
```

Рисунок 2.7 - Псевдокод алгоритму оцінки стану і пошук аномалій

Наступним кроком є алгоритм «моніторинг → рекомендація налаштування». Його роль полягає в формуванні рекомендацій на основі спостережуваних симптомів. Якщо модуль бачить високий p95 та p99, а також ознаки переповнення черг або довгих буферів, він може рекомендувати перехід на fq codel або перевірку його параметрів target та interval. Якщо основним симптомом є ріст retransmits і нестабільний throughput, модуль може рекомендувати перевірити алгоритм контролю перевантаження, наприклад tcp_congestion_control, а також оцінити доцільність ECN. Якщо проблема проявляється як надмірна затримка встановлення з’єднання і видно перевантаження при встановленні зєднань, можлива рекомендація перевірити somaxconn або tcp_max_syn_backlog. Якщо навпаки

видно, що потоки недоотримують пропускну здатність при невисоких втратах, можна перевірити параметри буферів tcp_rmem і tcp_wmem. Всі ці рекомендації повинні бути керованими правилами, тобто формуватися на основі чітко визначених умов. Модуль не повинен змінювати кілька параметрів одночасно. Він дивиться на тип проблеми, підбирає невелику множину релевантних параметрів і формує пояснення, чому саме вони запропоновані.

Безпечне застосування налаштувань є окремою частиною алгоритму (рисунок 2.8).

```
Початок

Отримати діагноз від модуля оцінки стану

Якщо виявлено queueing delay
    Запропонувати fq_codel або перевірку target та interval

Якщо виявлено високі retransmits
    Запропонувати перевірку tcp_congestion_control
    Запропонувати перевірку ECN і буферів tcp_rmem tcp_wmem

Якщо виявлено високу connect latency
    Запропонувати перевірку somaxconn та tcp_max_syn_backlog

Якщо режим = dry run
    Лише показати рекомендацію
    Завершити алгоритм

Зберегти поточні значення параметрів
Застосувати тільки одну зміну
Запустити контрольне вікно спостереження

Після завершення контрольного вікна
    Якщо p95 зменшився і retransmits не зросли і cpu не вийшов за межу
        Підтвердити нове налаштування
    Інакше
        Виконати rollback
        Повернути попереднє значення

Кінець
```

Рисунок 2.8 - Псевдокод алгоритму формування рекомендацій щодо налаштування з попередньою перевіркою та відкатом змін

Спочатку використовується режим пробного запуску без внесення змін. В цьому режимі агент нічого не змінює, а просто показує, які параметри можна змінити, з якої причини і який ефект очікується. Якщо оператор підтверджує дію або система працює в напівавтоматичному режимі, агент зберігає поточні значення параметрів і застосовує лише одну зміну. Далі протягом контрольного вікна

збираються ті самі метрики, що були на вході. Якщо p95 зменшується, retransmits не ростуть, а CPU не виходить за допустиму межу, зміна вважається корисною. Якщо ж після застосування налаштування латентність погіршується, throughput падає або CPU росте, виконується rollback і повертається попереднє значення.

Така схема дає змогу уникнути небезпечних автоматичних рішень і добре підходить для експлуатаційних середовищ, де навіть правильна на вигляд рекомендація може бути невдалою через специфіку конкретного навантаження.

2.3 Інформаційне забезпечення модуля

Вхідна інформація для модуля формується з подій ядра та коротких описів мережевих потоків, які доступні через eBPF-точки підключення. Сучасний eBPF підхід дозволяє зібрати значну частину мережевої телеметрії без повного дампу пакетів, перенести частину аналізу в ядро і передати назовні вже більш компактні результати. Крім того eBPF дає можливість будувати стабільний конвеєр збору та експорту метрик, а також інтегрувати його з Prometheus і Grafana.

Для мережевого моніторингу це означає, що базовим входом є не сирий повний трафік, а короткі події з потрібними полями.

До таких полів належать часова мітка, ідентифікатор процесу, номер мережевого інтерфейсу, адреси та порти джерела і призначення, назва або код протоколу, а також значення, пов’язані з поточним станом TCP чи UDP потоку. Для TCP це можуть бути rtt, retrans, cwnd, bytes, queue len, ознаки переходу між станами, а також допоміжні параметри для кореляції.

Якщо робота буде йти через tc або XDP частина полів надходить у вигляді пакетної телеметрії, а якщо через tracepoints або kprobes частина полів прив’язується до сокета чи події ядра.

Часові мітки і короткий стан потоку зберігаються в хеш мапах, а користувацький простір має отримувати вже обчислений або майже завершений результат, наприклад RTT. В середині ядра це може бути тимчасова службова

структура з ключем потоку та часовою міткою. На виході це буде окремий запис про вимірювання з коротким набором полів.

Вихідна інформація модуля складається з кількох класів результатів (рисунок 2.9).

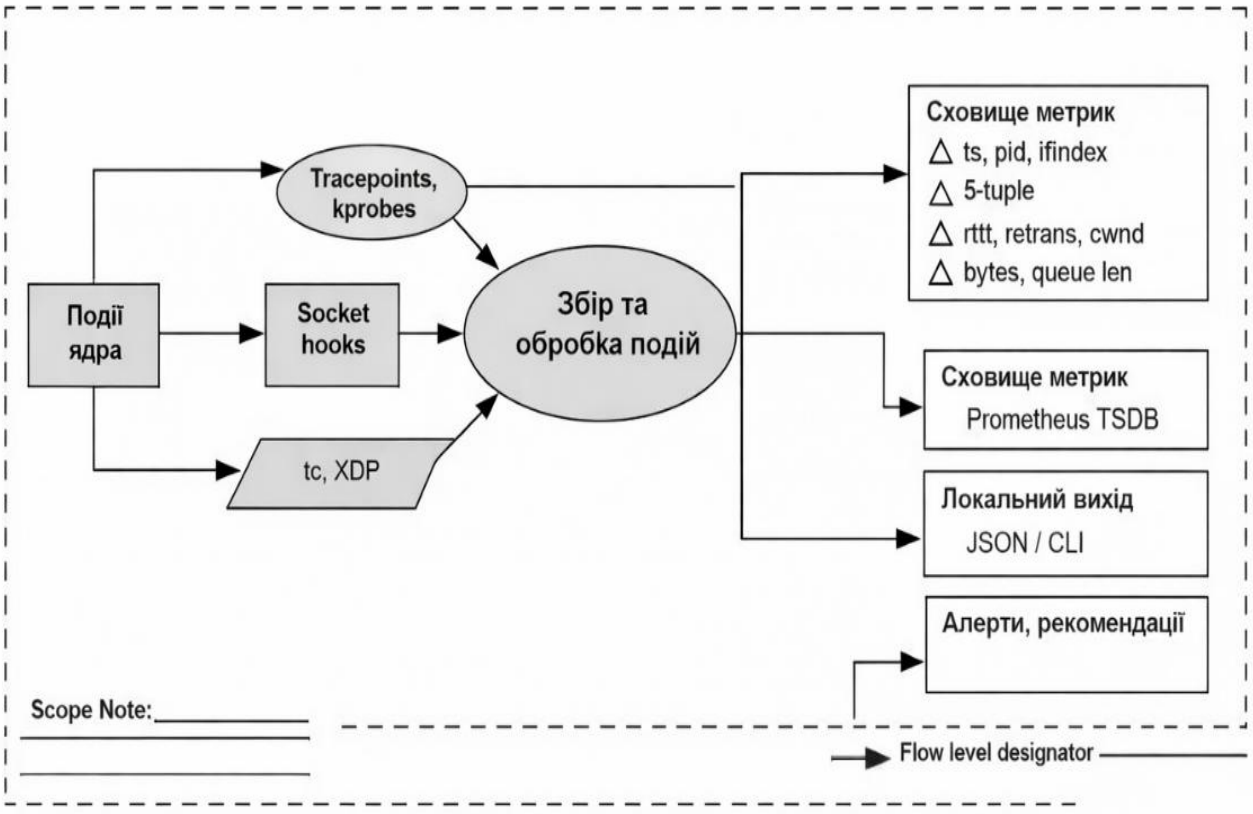


Рисунок 2.9 - Вхідна та вихідна інформація модуля моніторингу

Перший клас це агреговані метрики, які підходять для безперервного моніторингу. Другий клас це локальні виходи у вигляді JSON або короткого лог запису, які потрібні для діагностики, тестування та аналізу інцидентів. Третій клас це компактні звіти для CLI. Четвертий клас це попередження та рекомендації, які формуються модулем оцінки стану та блоком tune recommendation. Після цього дані передаються до шару візуалізації та використовуються для подальшого моніторингу.

Концептуальна модель даних (рисунок 2.10) потрібна для того, щоб перейти від сирих подій до зрозумілих сутностей предметної області. В цьому модулі центральною сутністю є Flow або Connection.

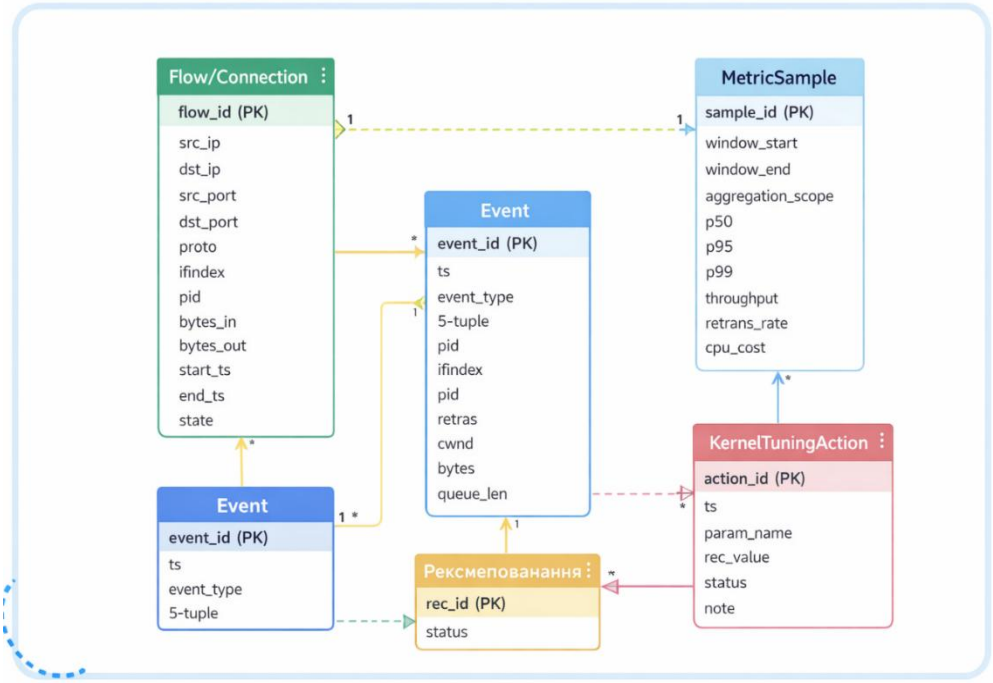


Рисунок 2.10 - Концептуальна модель даних модуля моніторингу

Потік або з’єднання поєднує адреси, порти, напрям, інтерфейс, процес, життєвий цикл і накопичені показники. На цьому рівні немає прив’язки до конкретної таблиці чи конкретного формату бази даних, а виділяються логічні об’єкти, з якими працює система.

Сутність Event описує окрему подію, що надійшла з ядра або була сформована на базі пакетної телеметрії. Для такої сутності атрибутами є event_id, ts, event_type, 5 tuple, pid, ifindex, а також значення, які характеризують подію, наприклад rtt, retrans, cwnd, bytes чи queue_len.

Event має короткий життєвий цикл і є носієм первинної інформації. Багато таких подій стосуються одного потоку. Між Event і Flow або Connection існує відношення багато до одного.

Сутність Flow або Connection є довшою за часом. Вона зберігає ключ потоку, часові межі життя, напрям, пов’язаний процес, інтерфейс та поточний агрегований стан. Ця сутність має такі поля, як flow_id, src_ip, dst_ip, src_port, dst_port, proto, pid, ifindex, start_ts, end_ts, bytes_in, bytes_out, state. Для TCP connection ця сутність фактично збігається із життєвим циклом з’єднання. Для UDP або QUIC це радше віконний опис потоку за п’ятипольним ключем.

Сутність MetricSample описує вже агрегований зріз за фіксований інтервал часу. Вона містить такі поля: sample_id, window_start, window_end, aggregation_scope, p50, p95, p99, throughput, retrans_rate, cpu_cost та інші похідні показники. Ця сутність є основою для подальшого експорту до TSDB і побудови дашбордів.

Сутність KernelTuningAction потрібна для підтримки алгоритму monitor → recommend tune. Вона зберігає інформацію про рекомендовану або виконану зміну параметра ядра. Типовими атрибутами є action_id, ts, target_scope, param_name, old_value, new_value, mode, status, rollback_flag, note.

Даталогічна модель (рисунок 2.11) відповідає на питання, в яких конкретних структурах зберігати інформацію після того, як концептуальні сутності вже визначені.

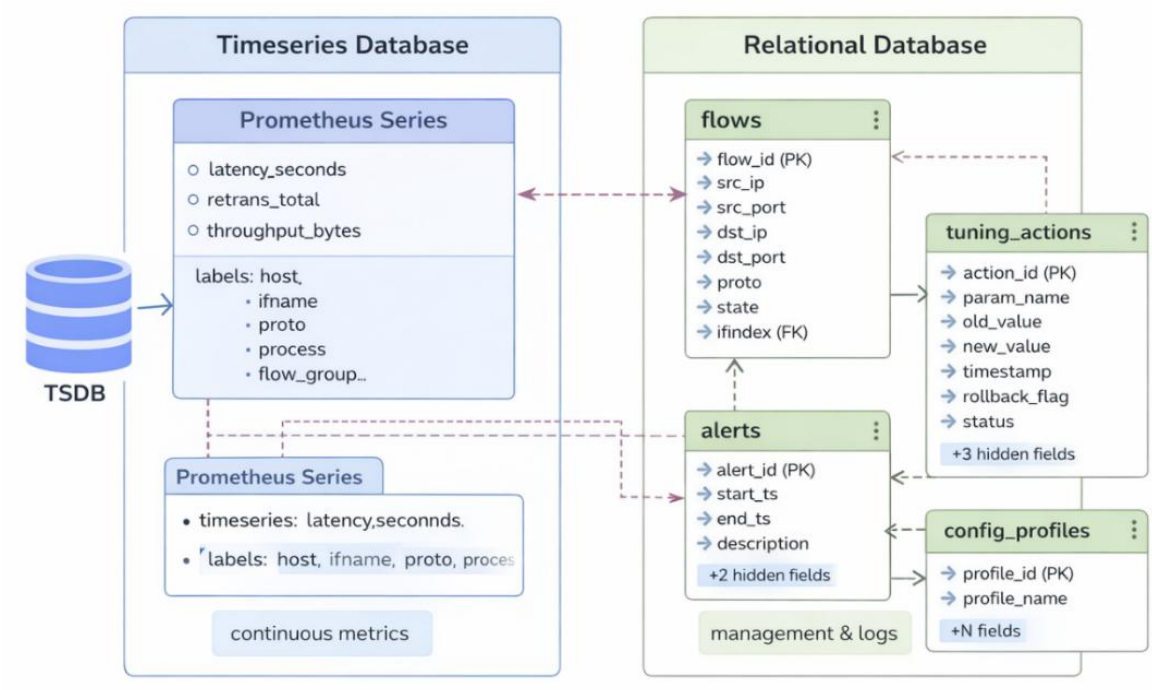


Рисунок 2.11 - Даталогічна модель з TSDB для метрик і реляційними таблицями для керування.

Для цього модуля застосовується змішаний підхід. Метрики часових рядів краще зберігати у TSDB або у форматі Prometheus series, а службові записи про

потоки, рекомендації, конфігурації та журнал змін мають бути в реляційній моделі або у легкому локальному сховищі.

Використання TSDB або Prometheus labels підходить для MetricSample, тому що ця сутність має часову вісь.

Якщо потрібно зберігати конфігурацію профілів, історію tune дій, прив'язку рекомендації до конкретного набору параметрів або дані для rollback, то зручніше використовувати реляційну схему. В такій схемі можуть бути таблиці flows, tuning_actions, alerts, config_profiles. Таблиця flows зберігатиме ідентифікатор потоку, 5 tuple, pid, ifindex, часові межі та базові показники. Таблиця tuning_actions зберігатиме назву параметра, старе і нове значення, статус виконання та ознаку відкату. Таблиця alerts потрібна для журналу деградацій та їхнього життєвого циклу. Таблиця config_profiles міститиме безпечні межі значень і підготовлені профілі параметрів.

Змішаний варіант полягає в тому, що TSDB і Prometheus зручні для швидких часових рядів, візуалізації та формування сповіщень, але вони не дуже зручні для зберігання журналу змін параметрів, причинно-наслідкових зв'язків та службової довготривалої інформації. Реляційна модель, навпаки, підходить для таких зв'язних записів, але не є оптимальною для швидкого зберігання великої кількості метрик по інтервалах. Тому комбінування цих двох підходів є найбільш раціональним.

3 ПРОГРАМНО-ТЕХНОЛОГІЧНЕ ЗАБЕЗПЕЧЕННЯ МОДУЛЯ МОНІТОРИНГУ НА БАЗІ eBPF

3.1 Реалізація програмного забезпечення модуля

Для частини яка буде працювати на ядрі було обрано мову C. Такий вибір обґрунтований тим, що eBPF-програми для Linux реалізуються саме цією мовою, оскільки вона підтримує роботу із заголовками vmlinux.h та бібліотекою libbpf. Крім того C дозволяє напряму описувати структури подій, працювати з допоміжними eBPF-функціями та реалізовувати мінімальну логіку безпосередньо в ядрі.

Для простору користувача була обрана мова Go для створення довготривалого агента, що одночасно виконує кілька завдань: завантажує eBPF-програму, читає події з кільцевого буфера, агрегує їх в часових вікнах, формує текстові звіти, віддає метрики через HTTP endpoint і записує знімки стану у JSON-файл.

Як базова бібліотека для взаємодії з eBPF в просторі користувача використано cilium/ebpf. Вона забезпечує завантаження згенерованих об’єктів, прикріплення програм до tracepoint, роботу з map-об’єктами та кільцевим буфером.

Як основне середовище виконання було використано Debian 13 з ядром Linux 6.12.

Програмна система побудована за дворівневим принципом. Нижній рівень працює в ядрі Linux і відповідає за первинний збір подій. Верхній рівень працює в просторі користувача і виконує керування модулем, агрегацію, експорт і візуалізацію.

Рівень ядра представлений eBPF-програмою monitor.bpf.c. Вона підключається до tracepoint/tcp/tcp_retransmit_skb і спрацьовує щоразу, коли в стеку TCP реєструється подія повторної передачі пакета. Програма зчитує часову мітку, ідентифікатор процесу, локальну і віддалену IPv4-адреси, локальний і віддалений порти, а також коротке ім’я поточного процесу. Далі ця інформація пакується в структуру події та передається у BPF_MAP_TYPE_RINGBUF.

Простір користувача представлений агентом, реалізованим мовою Go. Його роботу можна поділити на такі функціональні блоки. Перший блок виконує завантаження eBPF-програми та прикріплення її до потрібної контрольної точки. Другий блок читає події з кільцевого буфера і перетворює сирі записи у зрозумілу для програми структуру. Третій блок виконує агрегацію по часових інтервалах. Четвертий формує локальний CLI-звіт. П’ятий експортує метрики у форматі Prometheus. Шостий записує інтервальні знімки стану у JSON-файл. Сьомий формує вбудовану веб-візуалізацію у вигляді сторінки dashboard.

Структурно проект організований у вигляді окремих каталогів (рисунок 3.1).

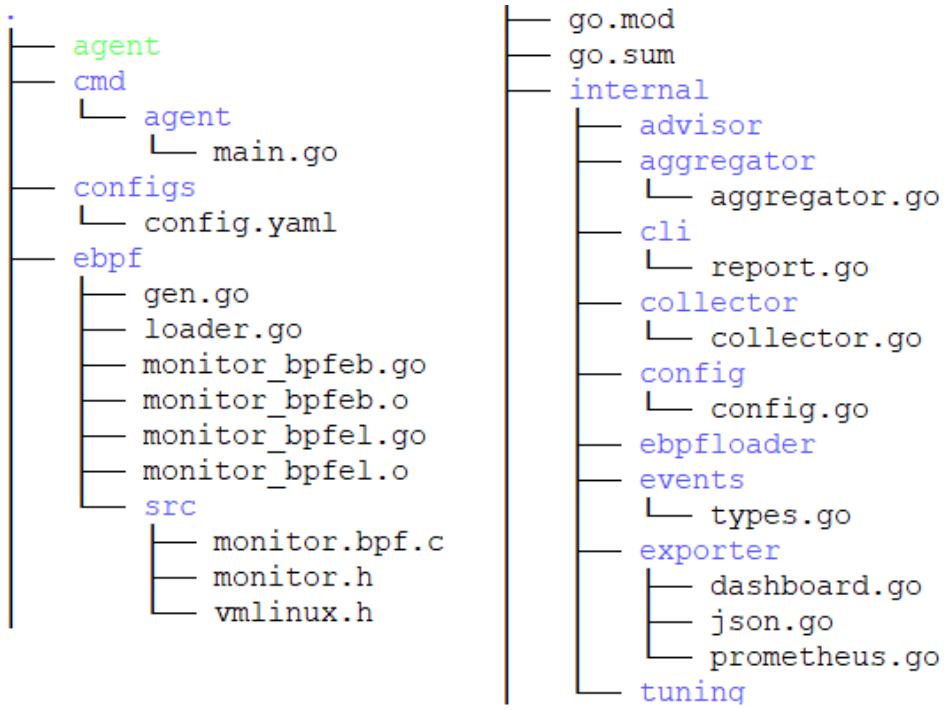


Рисунок 3.1 – Дерево каталогів проекту

В каталозі ebpf розміщуються eBPF-об’єкти, генератор і завантажувач. У каталозі internal винесено логіку колектора, агрегатора, конфігурації, експортерів і побудови CLI-звіту. В каталозі configs міститься конфігураційний файл, а в каталозі testdata — JSON snapshots, які утворюються під час виконання модуля.

Частину модуля в ядрі становить програма, яка відстежує факт повторної передачі TCP-сегмента. Для першого робочого варіанта було обрано цей тип події, тому що ретрансляція є одним із характерних проявів деградації мережевої

передачі і легко піддається експериментальній перевірці. Під час запуску тестового сценарію з використанням tc netem loss 20% модуль почав фіксувати події retransmit для процесів python3, curl та для службових потоків ядра.

В межах eBPF-програми було введено структуру tcp_retransmit_event, що містить наступні поля: час, ідентифікатор процесу, адреси, порти та ім’я процесу.

Передавання подій у простір користувача виконується через кільцевий буфер. Це дозволяє уникнути надмірного копіювання великих обсягів інформації та організувати ефективний механізм доставки компактних записів від ядра до агента. Після генерації заголовка vmlinux.h і об’єктних файлів було виконано інтеграцію eBPF-частини з Go-агентом через bpf2go.

Користувацький агент реалізований як окремий багатопотоковий процес. Його точкою входу є файл main.go, який запускає всі основні компоненти. На початку роботи агент читає конфігурацію, завантажує eBPF-модуль, відкриває кільцевий буфер подій і ініціалізує засоби експорту. Після цього він переходить у робочий режим, в якому працюють окремі паралельні потоки виконання.

Перший потік забезпечує безперервне читання подій з кільцевого буфера. Для цього використовується окремий колектор, який отримує сирі записи і декодує їх у структуру TCPRetransmitEvent.

Другий потік відповідає за інтервальну агрегацію. Вона працює за таймером і кожні п’ять секунд формує знімок поточного стану. На цьому етапі накопичені події групуються за двома ознаками: за процесом і за потоком.

Третій потік запускає HTTP-сервер. Через нього агент віддає метрики у форматі Prometheus на шляху /metrics та веб-сторінку на шляху /dashboard.

Окремий блок користувацького агента відповідає за JSON exporter. Він записує інтервальні знімки у файл ./testdata/snapshots.jsonl. Кожен запис містить часову мітку, загальну кількість подій, статистику за процесами та статистику за потоками.

Для фіксації всіх параметрів було додано конфігураційний файл config.yaml. В ньому визначаються інтервал агрегації, адреса HTTP-сервера метрик і шлях до

JSON-файлу. Для поточної реалізації був встановлений інтервал 5s, адреса :2112 і шлях ./testdata/snapshots.jsonl.

Окремо було створено інтерфейси подання результатів. Першим є CLI-звіт, який призначений для локального тестування, швидкої перевірки працездатності та діагностики на стенді. Другим є Prometheus endpoint для інтеграції з зовнішніми системами спостереження і дає стандартний машинний формат доступу до метрик. Третім є JSON snapshots і слугує для збереження історії тестів, подальшої обробки або вставлення фрагментів у пояснювальну записку. Четвертим є вбудований dashboard, що являє собою веб-сторінку, яка автоматично оновлюється і показує кількість retransmit за останній інтервал, топ процесів і топ потоків.

3.2 Інтерфейс взаємодії з користувачем та засоби подання результатів

Для користувача взаємодія з модулем організована як набір кількох взаємодоповнюючих способів подання результатів. Так як система орієнтована на технічного спеціаліста, вона повинна підтримувати кілька режимів роботи. Для цього було реалізовано швидкий текстовий звіт у терміналі, машинно-читабельний формат для систем моніторингу та візуальне подання поточного стану через браузер. В модулі реалізовано кілька взаємодоповнюючих інтерфейсів: CLI-режим, HTTP endpoint для Prometheus, JSON-знімки та вбудовану веб-панель.

Першим і найпростішим інтерфейсом взаємодії є командний режим. Він використовується для локальної діагностики, перевірки працездатності модуля та швидкого спостереження за поточним станом без запуску зовнішніх систем. Після старту агент виводить службові повідомлення про успішне завантаження модуля та переходить у режим періодичного формування зведеного звіту. В цьому режимі кожні п’ять секунд формується текстовий блок, який містить загальну кількість зафіксованих TCP retransmit подій за останній інтервал, розподіл за процесами та розподіл за потоками.

CLI-інтерфейс побудовано так, щоб користувач отримував лише основну інформацію без надмірної деталізації. Замість виведення кожної окремої події у

звичайному режимі агент формує коротке зведення, придатне для оперативної оцінки ситуації.

Другим інтерфейсом є HTTP endpoint /metrics, який реалізовано у форматі Prometheus (рисунок 3.2).

У поточній версії модуль формує три основні метрики.

```
root@Debain-oso:~/ebpf-monitor# curl -s http://127.0.0.1:2112/metrics | head -n 40
# HELP ebpf_monitor_tcp_retransmit_total Number of TCP retransmit events in the last aggrega
tion interval
# TYPE ebpf_monitor_tcp_retransmit_total gauge
ebpf_monitor_tcp_retransmit_total 0
# HELP ebpf_monitor_tcp_retransmit_by_process Number of TCP retransmit events in the last ag
gregation interval grouped by process
# TYPE ebpf_monitor_tcp_retransmit_by_process gauge
ebpf_monitor_tcp_retransmit_by_process{process="none"} 0
# HELP ebpf_monitor_tcp_retransmit_by_flow Number of TCP retransmit events in the last aggre
gation interval grouped by flow
# TYPE ebpf_monitor_tcp_retransmit_by_flow gauge
ebpf_monitor_tcp_retransmit_by_flow{flow="none"} 0
root@Debain-oso:~/ebpf-monitor#
```

Рисунок 3.2 – Представлення метрик модуля у форматі Prometheus

Перша показує загальну кількість retransmit подій за останній інтервал. Друга метрика подає ці події в розрізі процесів. Третя подає їх у розрізі потоків. Формат відповіді сумісний зі стандартом Prometheus exposition format, тому такі метрики можуть бути безпосередньо зчитані зовнішнім збирачем.

Крім того зовнішній моніторинг не потребує спеціального доступу до внутрішньої логіки модуля. Можна використовувати стандартний HTTP-запит до порту, вказаного в конфігурації.

Третім способом подання результатів є запис інтервальних знімків у JSON-файл. Такий інтерфейс орієнтований на локальне збереження даних, подальшу автоматизовану обробку, архівування результатів тестів. JSON snapshots дозволяють зберігати послідовність інтервальних записів у файлі і повертатися до них пізніше. Це робить цей інтерфейс корисним для документування експериментів.

В реалізованому модулі кожен запис містить часову мітку, загальну кількість подій за інтервал, статистику за процесами та статистику за потоками.

Четвертим інтерфейсом є вбудована веб-сторінка /dashboard. Вона була додана як засіб візуалізації. Це дозволяє швидко продемонструвати роботу модуля через браузер. Сторінка dashboard автоматично оновлюється кожні п’ять секунд і містить три основні блоки.

В першому блоці показується кількість TCP retransmit за останній інтервал. У другому блоці відображаються процеси, які найчастіше фіксувалися у знімку. У третьому блоці відображаються потоки з найбільшою кількістю retransmit-подій.

На відміну від текстового CLI-звіту, dashboard придатний для скріншотів, презентації результатів та швидкого візуального огляду без аналізу консольного логу.

3.3 Тестування та аналіз результатів роботи модуля

Після реалізації програмного модуля було проведено його практичне тестування в середовищі Linux.

Як тестове середовище було використано Debian 13 з ядром Linux 6.12. Реалізація працювала у віртуальній машині, що дало змогу контролювати середовище виконання та поступово перевіряти роботу кожного компонента. Для генерації мережевого трафіку застосовувався локальний HTTP-сервер на базі python3 -m http.server, який роздавав тестовий файл. Роль клієнта виконував curl, а для створення штучного погіршення каналу на інтерфейсі lo застосовувався механізм tc netem loss 20%. Цей сценарій дозволив отримати керовані TCP retransmit-події, яких було достатньо для перевірки роботи модуля

Під час підготовки тестового стенда було підтверджено коректне завантаження eBPF-програми, прикріплення її до tracepoint/tcp/tcp_retransmit_skb, а також доступність користувацького агента, який запускав CLI-звіт, Prometheus endpoint, JSON snapshots і веб-сторінку dashboard. Це дозволило виконувати перевірку захоплення подій та правильності роботи всіх засобів подання результатів.

Спочатку було перевірено базову працездатність модуля. На цьому етапі перевіряли, чи агент успішно запускається, завантажує eBPF-об’єкт, відкриває кільцевий буфер, створює HTTP-сервер і переходить в робочий режим. Після запуску в журналі відображалися службові повідомлення про успішне завантаження модуля (рисунок 3.3), адресу http://127.0.0.1:2112/metrics, адресу http://127.0.0.1:2112/dashboard, шлях до JSON snapshots і поточний інтервал агрегації 5s.

```
root@Debain-oso:~/ebpf-monitor# ./agent
2026/05/30 08:23:13 eBPF monitor loaded
2026/05/30 08:23:13 collecting TCP retransmit events, press Ctrl+C to stop
2026/05/30 08:23:13 Prometheus metrics available at http://127.0.0.1:2112/metrics
2026/05/30 08:23:13 Dashboard available at http://127.0.0.1:2112/dashboard
2026/05/30 08:23:13 JSON snapshots will be written to ./testdata/snapshots.jsonl
2026/05/30 08:23:13 aggregation interval: 5s
```

Рисунок 3.3 – Запуск агента та службові повідомлення про ініціалізацію модуля

Наступною перевіркою був моніторинг сирих подій retransmit. Під час ранніх тестів модуль виводив записи виду tcp_retransmit pid=... comm=... saddr=... daddr=... sport=... dport=..., що підтвердило правильність декодування структури події та відповідність адрес і портів реальному трафіку.

Окремо було перевірено коректність завершення роботи модуля. Після натискання Ctrl+C агент завершувався без аварій і закривав усі активні компоненти.

Основним етапом тестування була перевірка інтервальної агрегації. Вона показує, чи здатний модуль перетворювати окремі сирі події на компактний і корисний звіт за фіксоване часове вікно. В реалізації тривалість вікна встановлена в 5 секунд. Кожен інтервал завершується формуванням знімка, який містить загальну кількість retransmit подій, розподіл за процесами та розподіл за потоками.

Під час роботи без тестового трафіку модуль сформував нульові знімки. У CLI відображалися записи total events: 0, а також no data.

Після ввімкнення tc netem loss 20% та запуску передавання файла за допомогою curl у зведеннях з’явилися ненульові значення. Наприклад, в одному з перших інтервалів було зафіксовано (рисунок 3.4) total events: 20, де серед процесів

були swapper/1, swapper/3, swapper/2 і ksoftirqd/1, а серед потоків було зафіксовано основний напрям 127.0.0.1:8080 -> 127.0.0.1:52940.

```
2026/05/30 08:27:39
=== TCP retransmit summary ===
total events: 6

by process:
  swapper/1: 2
  swapper/2: 2
  ksoftirqd/2: 1
  swapper/0: 1

by flow:
  127.0.0.1:51830 -> 127.0.0.1:8080: 3
  127.0.0.1:8080 -> 127.0.0.1:51830: 3
2026/05/30 08:27:44
```

Рисунок 3.4 – Консольний зведений звіт модуля під час появи TCP retransmit подій

В інших інтервалах спостерігалися значення 8, 4, 3, 15 та інші, що показує реальний змінний характер retransmit під час тесту.

Після завершення тестового навантаження модуль повернувся до нульового стану. В наступних часових вікнах знову з’являлися інтервали з total events: 0, що означає правильну роботу механізму SnapshotAndReset. Це підтверджує, що система не накопичує події нескінченно, а відображає стан лише за поточний часовий інтервал.

Після перевірки CLI-звіту було протестовано інші інтерфейси подання результатів. Першим із них став Prometheus endpoint /metrics. На нульовому стані він повертав ebpf_monitor_tcp_retransmit_total 0, а також нульові значення для групування за процесами і потоками. Цей endpoint може бути використаний як вхід для зовнішнього збирача метрик і побудови графіків.

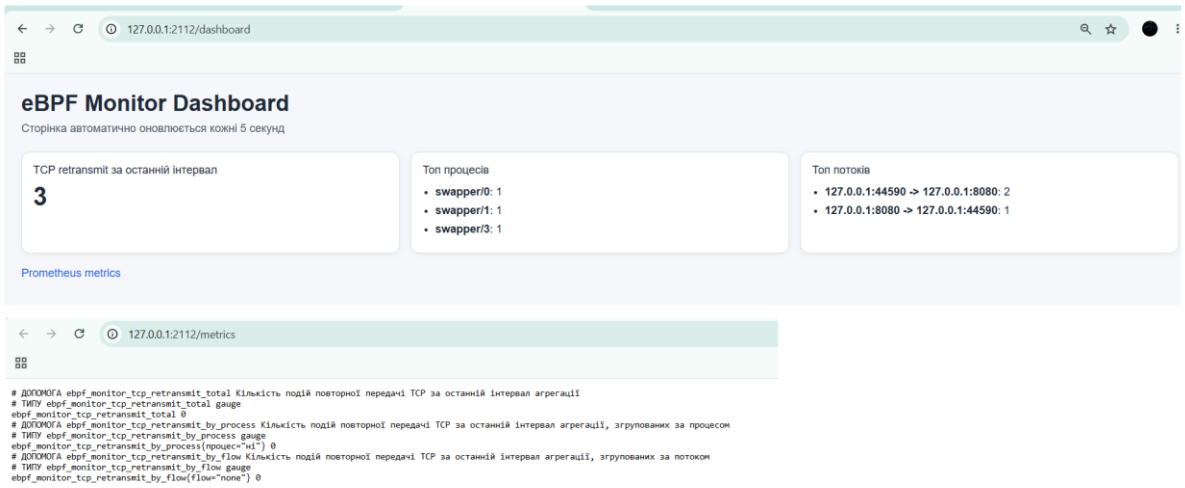
Другим інтерфейсом був JSON exporter (рисунок 3.5). Файл ./testdata/snapshots.jsonl містив часові знімки з полями timestamp, total_events, by_process та by_flow.


```
{
  "timestamp": "2026-05-30T08:43:59Z",
  "total_events": 1,
  "by_process": [
    {
      "Comm": "python3",
      "Count": 1
    }
  ],
  "by_flow": [
    {
      "Flow": "127.0.0.1:8080 -\u003e 127.0.0.1:34388",
      "Count": 1
    }
  ]
}
```


Рисунок 3.5 – Приклад інтервальних JSON snapshots, сформованих модулем

Під час відсутності мережевого навантаження у файлі з’являлися записи з total_events: 0, а під час тесту з retransmit були зафіксовані ненульові знімки, наприклад із total_events: 8, де поле by_process містило значення для swapper/0, swapper/3 і swapper/1, а у by_flow — головний потік 127.0.0.1:8080 -> 127.0.0.1:49192. JSON snapshots є узгодженими з CLI-звітом і можуть використовуватися для збереження та повторного аналізу результатів.

Третім інтерфейсом став вбудований dashboard (рисунок 3.6). Через запит до http://127.0.0.1:2112/dashboard було підтверджено, що агент віддає HTML-сторінку з автоматичним оновленням.



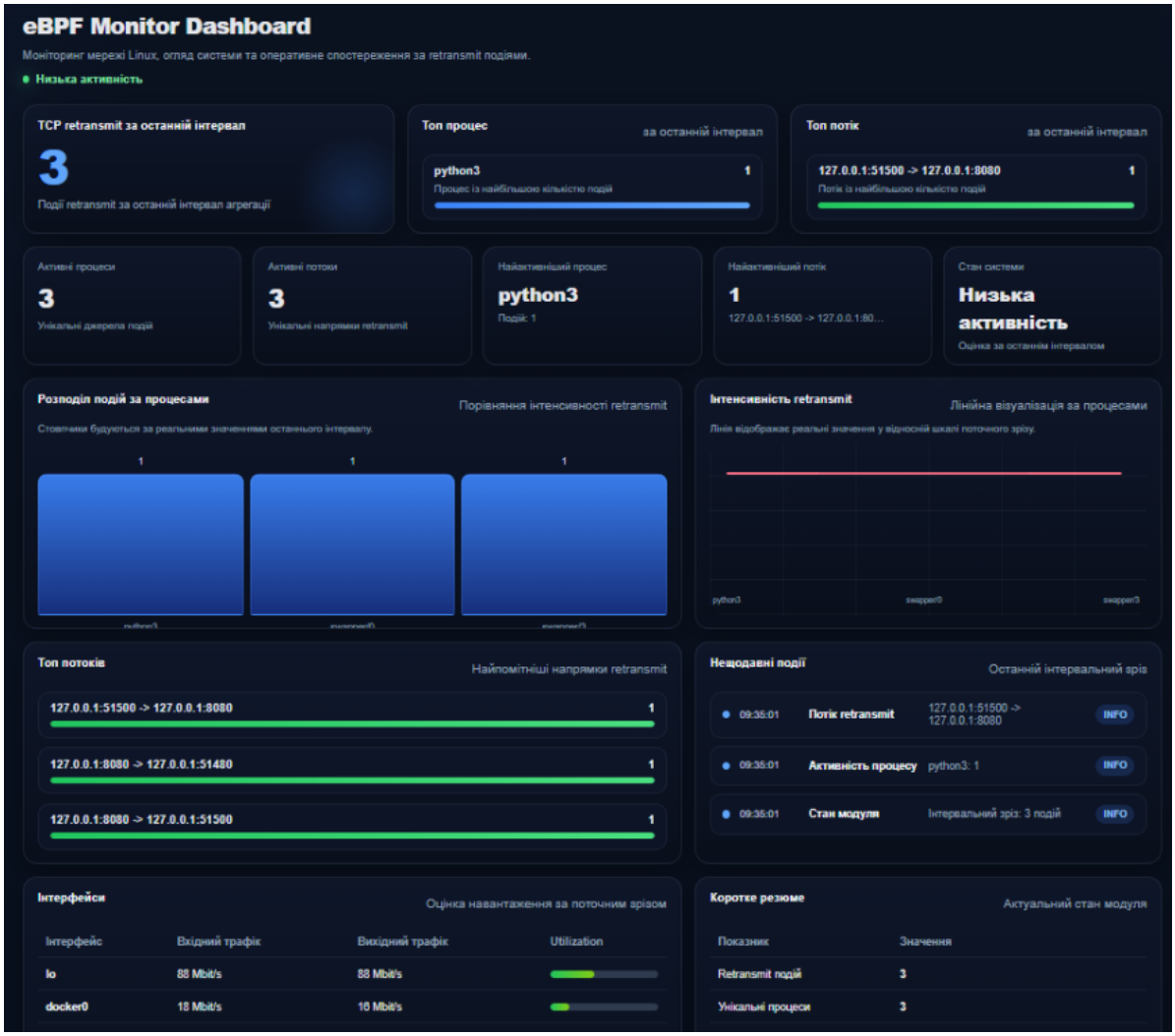


Рисунок 3.6 – Вбудована веб-візуалізація стану модуля у вигляді dashboard

В поточному вигляді сторінка містить три основні блоки: кількість retransmit за останній інтервал, топ процесів та топ потоків.

ВИСНОВКИ

При виконанні кваліфікаційної роботи було досягнуто таких результатів:

1. Проведено аналіз предметної області мережевої телеметрії в Linux, розглянуто особливості flow-телеметрії, роль метрик latency, throughput, retransmissions, queueing delay та CPU cost, а також показано значення постійного спостереження за мережевим стеком у сучасних серверних і хмарних середовищах.

2. Проаналізовано основні інструменти моніторингу мережевого стеку Linux, такі як tcpdump, ss, iproute2, perf, ftrace, BCC та bpftrace. Визначено, що класичні засоби корисні для діагностики та разових досліджень, але для безперервного моніторингу більш доцільним є використання eBPF, який може збирати події безпосередньо в ядрі з меншими накладними витратами.

3. Сформульовано постановку задачі розроблення програмного модуля моніторингу на базі eBPF, який забезпечує безперервний збір телеметрії мережевого стеку Linux.

4. Розроблено архітектуру модуля, побудовану за дворівневим принципом. Нижній рівень реалізовано у вигляді eBPF-програм у ядрі Linux, а верхній рівень — у вигляді користувацького агента, який виконує приймання подій, агрегацію, експорт і візуалізацію даних.

5. Розроблено алгоритмічне забезпечення модуля, що включає алгоритми збору TCP latency, обліку retransmit-подій, базової QUIC-телеметрії на рівні UDP та формування рекомендацій щодо налаштування параметрів мережевого стеку.

6. Розроблено інформаційне забезпечення модуля, визначено структури подій, формат агрегованих метрик і логіку формування сутностей Event та Flow.

7. Проведено вибір програмно-технологічних засобів реалізації. Для ядра вибрано мову C, для користувацького агента — Go, для взаємодії з eBPF у просторі користувача — бібліотеку cilium/ebpf.

8. Реалізовано програмне забезпечення модуля, яке включає eBPF-програму, механізм передавання подій, користувацький агент для декодування та агрегації подій, а також модулі експорту й подання результатів.

9. Реалізовано інтерфейси взаємодії з користувачем і зовнішніми системами моніторингу, зокрема CLI-звіт, Prometheus endpoint, JSON snapshots та веб-панель dashboard.

10. Проведено тестування працездатності модуля. Для генерації контрольованих retransmit-подій використано локальний HTTP-сервер, клієнт curl та механізм tc netem loss 20%.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. NAPI. Linux kernel documentation. URL: https://docs.kernel.org/networking/napi.html (дата звернення: 09.03.2026).
2. Høiland-Jørgensen T., Brouer J. D., Borkmann D., Fastabend J., Herbert T., Ahern D., Miller D. The eXpress Data Path: fast programmable packet processing in the operating system kernel // *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*. 2018. P. 54–66.
3. Nichols K., Jacobson V. Controlling queue delay // *Communications of the ACM*. 2012. Vol. 55, No. 7. P. 42–50.
4. tc-fq_codel(8) — Linux manual page. URL: https://man7.org/linux/man-pages/man8/tc-fq_codel.8.html (дата звернення: 09.03.2026).
5. Cardwell N., Cheng Y., Gunn C. S., Yeganeh S. H., Jacobson V. BBR: congestion-based congestion control // *Communications of the ACM*. 2017. Vol. 60, No. 2. P. 58–66.
6. Hock M., Bless R., Zitterbart M. Experimental evaluation of BBR congestion control // *2017 IEEE 25th International Conference on Network Protocols*. 2017. P. 1–10.
7. Dean J., Barroso L. A. The tail at scale: software techniques that tolerate latency variability are vital to building responsive large-scale web services // *Communications of the ACM*. 2013.
8. Rüth J., Poese I., Dietzel C., Hohlfeld O. A first look at QUIC in the wild // International Conference on Passive and Active Network Measurement. *Cham*, 2018. P. 255–268.
9. RFC 9002. QUIC Loss Detection and Congestion Control. URL: https://datatracker.ietf.org/doc/html/rfc9002 (дата звернення: 09.03.2026).
10. tcpdump. GitHub repository. URL: https://github.com/the-tcpdump-group/tcpdump (дата звернення: 09.03.2026).
11. McCanne S., Jacobson V. The BSD packet filter: a new architecture for user-level packet capture // *USENIX Winter Conference*. 1993. P. 259–270.

12. Ковальов О. О. Клієнт-серверний додаток моніторингу мережевого трафіку домашньої мережі : *кваліфікаційна робота*. 2025.

13. ss(8). Linux manual page. URL: https://linux.die.net/man/8/ss (дата звернення: 09.03.2026).

14. perf tutorial. PerfWiki. URL: https://perfwiki.github.io/main/tutorial/ (дата звернення: 09.03.2026).

15. Tracing. Linux kernel documentation. URL: https://docs.kernel.org/trace/index.html (дата звернення: 09.03.2026).

16. Weaver V. M. Linux perf_event features and overhead // The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, *FastPath*. 2013. P. 5.

17. Cassagnes C., Trestioreanu L., Joly C., State R. The rise of eBPF for non-intrusive performance monitoring // *NOMS 2020* – 2020 IEEE/IFIP Network Operations and Management Symposium. 2020. P. 1–7.

18. BCC. GitHub repository. URL: https://github.com/iovisor/bcc (дата звернення: 09.03.2026).

19. Sundberg S., Brunstrom A., Ferlin-Reiter S., Høiland-Jørgensen T., Brouer J. D. Passive monitoring of network latency at high line rates // *17th Swedish National Computer Networking Workshop*. Stockholm, 2022.

20. Hinz J. T., Addanki V., Györgyi C., Jepsen T., Schmid S. TCP's third eye: leveraging eBPF for telemetry-powered congestion control // *Proceedings of the 1st Workshop on eBPF and Kernel Extensions*. 2023. P. 1–7.

21. Miano S., Chen X., Basat R. B., Antichi G. Fast in-kernel traffic sketching in eBPF // *ACM SIGCOMM* Computer Communication Review. 2023. Vol. 53, No. 1. P. 3–13.

22. Craun M., Hussain K., Gautam U., Ji Z., Rao T., Williams D. Eliminating eBPF tracing overhead on untraced processes // *Proceedings of the ACM SIGCOMM 2024* Workshop on eBPF and Kernel Extensions. 2024. P. 16–22.

23. Tsubouchi Y., Furukawa M., Matsumoto R. Low overhead TCP/UDP socket-based tracing for discovering network services dependencies // Journal of Information Processing. 2022. Vol. 30. P. 260–268.

24. De Coninck Q., Navarre L., Rybowski N. On integrating eBPF into pluginized protocols // *ACM SIGCOMM Computer Communication Review*. 2024. Vol. 53, No. 3. P. 2–8.

25. Tian W., Li Y., Zhao J., Wu S., Pan J. An eBPF-based trace-driven emulation method for satellite networks // *IEEE Networking Letters*. 2024. Vol. 6, No. 3. P. 188–192.

26. Хрущак С., Бойко О., Терьохіна М. Безпековий моніторинг інформаційних систем за допомогою eBPF // *Наука і техніка сьогодні*. 2024. № 4(32). С. 1251–1262. DOI: 10.52058/2786-6025-2024-4(32)-1251-1262.

27. Бойко Д. Д. Застосунок моніторингу трафіку та визначення DDoS-атак за допомогою eBPF : кваліфікаційна робота. 2023.

28. Журавчак Д. Ю. Моніторинг вірусів-вимагачів за допомогою розширеного Берклійського пакетного фільтра eBPF та машинного навчання // Science-Based Technologies. 2023. Т. 60, № 4.

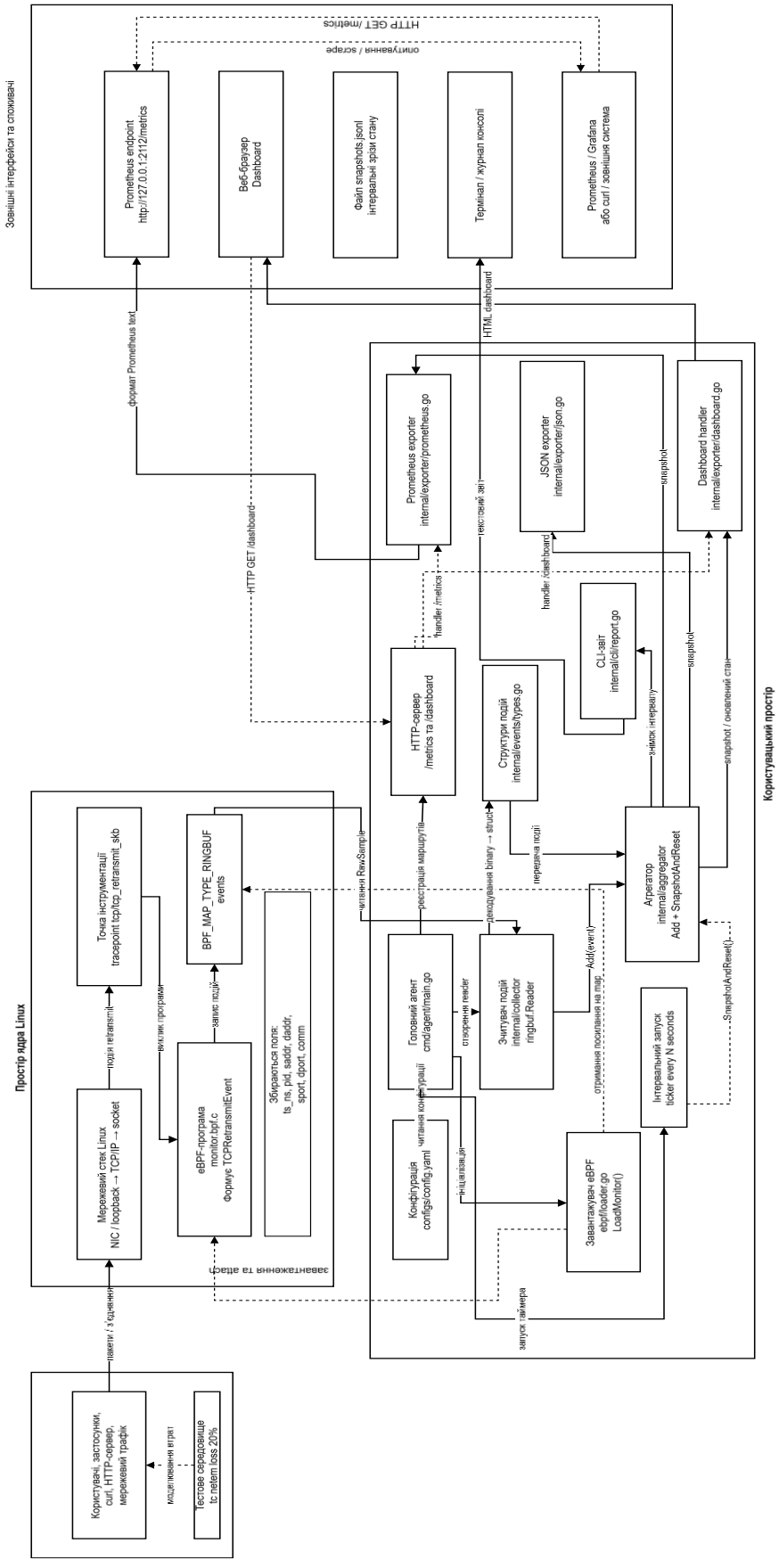
29. PerfWiki. URL: https://perfwiki.github.io/main/ (дата звернення: 09.03.2026).

30. Комар М.П., Саченко А.О., Васильків Н.М., Гладій Г.М., Коваль В.С., Ліп’яніна-Гончаренко Х.В. Методичні рекомендації до виконання кваліфікаційної роботи з освітньо-професійної програми «Комп’ютерні науки» спеціальності 122 «Комп’ютерні науки» за першим (бакалаврським) рівнем вищої освіти. Тернопіль: ЗУНУ, 2024. 52 с.

31. Цимбалюк А. М. Програмний модуль моніторингу на базі eBPF для підвищення мережевої продуктивності Linux. Традиційні та інноваційні підходи до наукових досліджень : матеріали XI Міжнар. наук.-практ. конф. (м. Луцьк, 19 черв. 2026 р.). Луцьк, 2026.

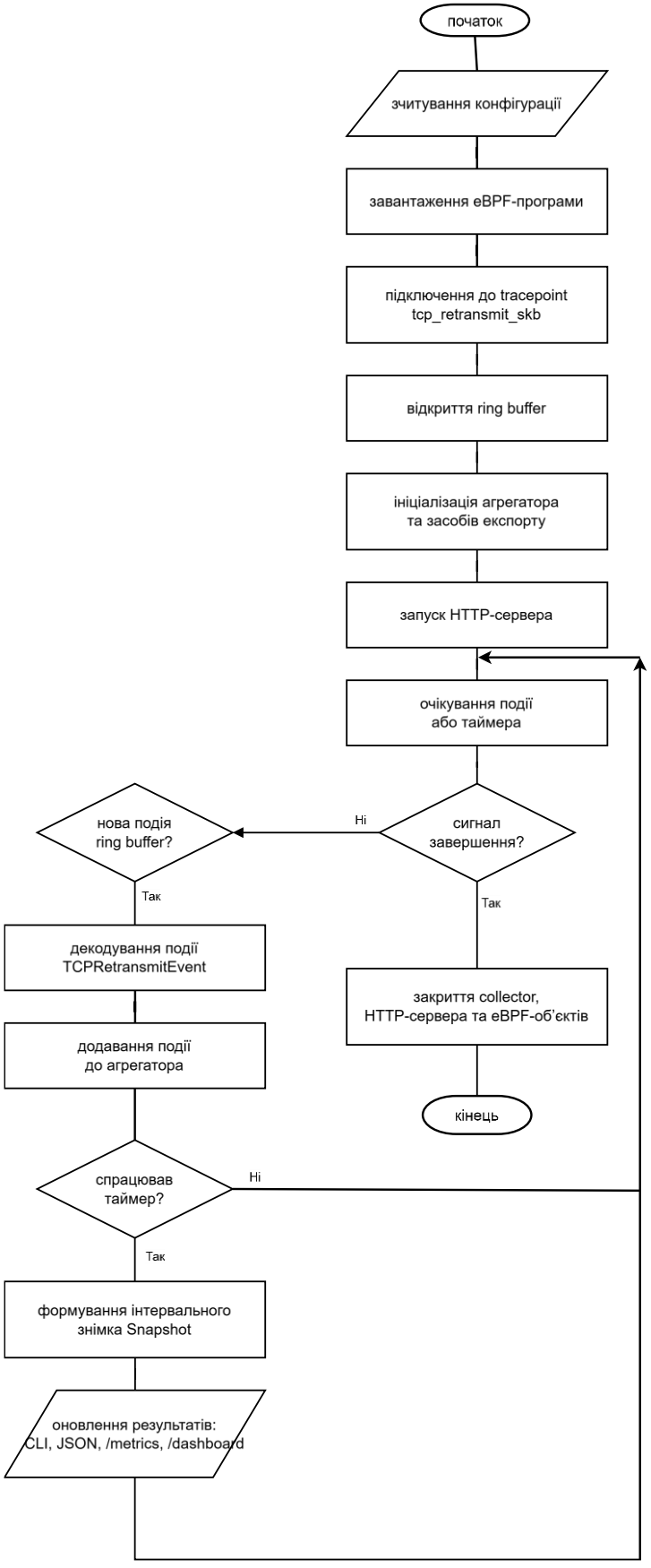
Додаток А

Архітектура модуля



Додаток Б

Блок-схема алгоритму роботи модуля моніторингу на базі eBPF



Додаток В

Код основних програмних компонент

```
==============================
configs/config.yaml
==============================
aggregation_interval_seconds: 5
metrics_address: ":2112"
json_output_path:
"./testdata/snapshots.jsonl"

==============
ebpf/gen.go
==============
package ebpf

//go:generate  /root/go/bin/bpf2go  -cc
clang  -cflags  "-O2  -g  -Wall"  monitor
./src/monitor.bpf.c -- -I./src

==============
ebpf/loader.go
==============
package ebpf

import (
      "fmt"

      "github.com/cilium/ebpf"
      "github.com/cilium/ebpf/link"
)

type Monitor struct {
      objs monitorObjects
      tp   link.Link
}

func LoadMonitor() (*Monitor, error) {
      var objs monitorObjects
      if              err              :=
loadMonitorObjects(&objs,  nil);  err  !=
nil {
            return nil, fmt.Errorf("load
monitor objects: %w", err)
      }

      tp, err := link.Tracepoint("tcp",
"tcp_retransmit_skb",
objs.HandleTcpRetransmit, nil)
      if err != nil {
            objs.Close()
            return                  nil,
fmt.Errorf("attach tracepoint: %w", err)
      }

      return &Monitor{
            objs: objs,
            tp:   tp,
      }, nil
}

func (m *Monitor) Close() error {
      if m == nil {
            return nil
      }

      if m.tp != nil {
            _ = m.tp.Close()
      }

      return m.objs.Close()
}

func (m *Monitor) EventsMap() *ebpf.Map
{
      return m.objs.Events
}

==============
ebpf/src/monitor.h
==============
#ifndef __MONITOR_H__
#define __MONITOR_H__

#define TASK_COMM_LEN 16

struct tcp_retransmit_event {
    __u64 ts_ns;
    __u32 pid;
    __u32 saddr;
    __u32 daddr;
    __u16 sport;
    __u16 dport;
    char comm[TASK_COMM_LEN];
};

#endif

==============
ebpf/src/monitor.bpf.c
==============
#include "vmlinux.h"
#include <bpf/bpf_core_read.h>
#include <bpf/bpf_endian.h>
#include <bpf/bpf_helpers.h>

#include "monitor.h"

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

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


SEC("tracepoint/tcp/tcp_retransmit_skb")
int         handle_tcp_retransmit(struct
trace_event_raw_tcp_event_sk_skb *ctx)
{
    struct tcp_retransmit_event *event;
    __u64 pid_tgid;
```

```
    event = bpf_ringbuf_reserve(&events,
sizeof(*event), 0);
    if (!event) {
        return 0;
    }

    event->ts_ns = bpf_ktime_get_ns();

    pid_tgid                                =
bpf_get_current_pid_tgid();
    event->pid = pid_tgid >> 32;

    event->saddr = ctx->saddr;
    event->daddr = ctx->daddr;
    event->sport     =     bpf_ntohs(ctx-
>sport);
    event->dport     =     bpf_ntohs(ctx-
>dport);

    bpf_get_current_comm(&event->comm,
sizeof(event->comm));

    bpf_ringbuf_submit(event, 0);
    return 0;
}
```

```
==============
internal/events/types.go
==============
package events

import (
      "encoding/binary"
      "net"
      "strings"
)

type TCPRetransmitEvent struct {
      TsNs  uint64
      Pid   uint32
      SAddr uint32
      DAddr uint32
      SPort uint16
      DPort uint16
      Comm  [16]byte
}

func (e TCPRetransmitEvent) CommString()
string {
      s := string(e.Comm[:])
      s = strings.TrimRight(s, "\x00")
      return strings.TrimSpace(s)
}

func        (e        TCPRetransmitEvent)
SAddrString() string {
      return ipv4FromUint32(e.SAddr)
}

func        (e        TCPRetransmitEvent)
DAddrString() string {
      return ipv4FromUint32(e.DAddr)
}

func ipv4FromUint32(v uint32) string {
      buf := make([]byte, 4)
      binary.LittleEndian.PutUint32(buf,
v)
      return net.IP(buf).String()
}
```

```
==============
internal/collector/collector.go
==============
package collector

import (
      "bytes"
      "encoding/binary"
      "fmt"

      "github.com/cilium/ebpf"
      "github.com/cilium/ebpf/ringbuf"

      "ebpf-monitor/internal/events"
)

type Collector struct {
      reader *ringbuf.Reader
}

func      New(eventsMap      *ebpf.Map)
(*Collector, error) {
      reader,           err           :=
ringbuf.NewReader(eventsMap)
      if err != nil {
            return                  nil,
fmt.Errorf("create  ring  buffer  reader:
%w", err)
      }

      return &Collector{reader: reader},
nil
}

func      (c      *Collector)      Read()
(events.TCPRetransmitEvent, error) {
      var                         event
events.TCPRetransmitEvent

      record, err := c.reader.Read()
      if err != nil {
            return event, err
      }

      if             err             :=
binary.Read(bytes.NewReader(record.RawSa
mple), binary.LittleEndian, &event); err
!= nil {
            return                event,
fmt.Errorf("decode event: %w", err)
      }

      return event, nil
}

func (c *Collector) Close() error {
      if c == nil || c.reader == nil {
```

```
            return nil
      }
      return c.reader.Close()
}


==============
internal/aggregator/aggregator.go
==============
package aggregator

import (
      "fmt"
      "sort"
      "sync"

      "ebpf-monitor/internal/events"
)

type Snapshot struct {
      TotalEvents int
      ByComm      []CommStat
      ByFlow      []FlowStat
}

type CommStat struct {
      Comm  string
      Count int
}

type FlowStat struct {
      Flow  string
      Count int
}

type Aggregator struct {
      mu     sync.Mutex
      total  int
      byComm map[string]int
      byFlow map[string]int
}

func New() *Aggregator {
      return &Aggregator{
            byComm:
make(map[string]int),
            byFlow:
make(map[string]int),
      }
}

func    (a    *Aggregator)    Add(event
events.TCPRetransmitEvent) {
      a.mu.Lock()
      defer a.mu.Unlock()

      a.total++

      comm := event.CommString()
      if comm == "" {
            comm = "unknown"
      }
      a.byComm[comm]++

      flow   :=   fmt.Sprintf("%s:%d   ->
%s:%d",
            event.SAddrString(),
            event.SPort,
            event.DAddrString(),
            event.DPort,
      )
      a.byFlow[flow]++
}

func (a *Aggregator) SnapshotAndReset()
Snapshot {
      a.mu.Lock()
      defer a.mu.Unlock()

      snapshot := Snapshot{
            TotalEvents: a.total,
            ByComm:     make([]CommStat,
0, len(a.byComm)),
            ByFlow:     make([]FlowStat,
0, len(a.byFlow)),
      }

      for comm, count := range a.byComm
{
            snapshot.ByComm            =
append(snapshot.ByComm, CommStat{
                  Comm:  comm,
                  Count: count,
            })
      }

      for flow, count := range a.byFlow
{
            snapshot.ByFlow            =
append(snapshot.ByFlow, FlowStat{
                  Flow:  flow,
                  Count: count,
            })
      }

      sort.Slice(snapshot.ByComm,
func(i, j int) bool {
            if  snapshot.ByComm[i].Count
== snapshot.ByComm[j].Count {
                  return
snapshot.ByComm[i].Comm                <
snapshot.ByComm[j].Comm
            }
            return
snapshot.ByComm[i].Count               >
snapshot.ByComm[j].Count
      })

      sort.Slice(snapshot.ByFlow,
func(i, j int) bool {
            if  snapshot.ByFlow[i].Count
== snapshot.ByFlow[j].Count {
                  return
snapshot.ByFlow[i].Flow                <
snapshot.ByFlow[j].Flow
            }
            return
snapshot.ByFlow[i].Count               >
snapshot.ByFlow[j].Count
```

```
	})

	a.total = 0
	a.byComm = make(map[string]int)
	a.byFlow = make(map[string]int)

	return snapshot
}


==============
internal/cli/report.go
==============
package cli

import (
	"fmt"
	"strings"

	"ebpf-monitor/internal/aggregator"
)

func BuildSnapshotReport(snapshot aggregator.Snapshot) string {
	var b strings.Builder

	fmt.Fprintf(&b, "=== TCP retransmit summary ===\n")
	fmt.Fprintf(&b, "total events: %d\n", snapshot.TotalEvents)

	fmt.Fprintf(&b, "\nby process:\n")
	if len(snapshot.ByComm) == 0 {
		fmt.Fprintf(&b, "  no data\n")
	} else {
		limit := min(5, len(snapshot.ByComm))
		for i := 0; i < limit; i++ {
			item := snapshot.ByComm[i]
			fmt.Fprintf(&b, "  %s: %d\n", item.Comm, item.Count)
		}
	}

	fmt.Fprintf(&b, "\nby flow:\n")
	if len(snapshot.ByFlow) == 0 {
		fmt.Fprintf(&b, "  no data\n")
	} else {
		limit := min(5, len(snapshot.ByFlow))
		for i := 0; i < limit; i++ {
			item := snapshot.ByFlow[i]
			fmt.Fprintf(&b, "  %s: %d\n", item.Flow, item.Count)
		}
	}

	return b.String()
}

func min(a, b int) int {
	if a < b {
		return a
	}
	return b
}


==============
internal/config/config.go
==============
package config

import (
	"bufio"
	"fmt"
	"os"
	"strconv"
	"strings"
)

type Config struct {
	AggregationIntervalSeconds int
	MetricsAddress             string
	JSONOutputPath             string
}

func Load(path string) (Config, error) {
	file, err := os.Open(path)
	if err != nil {
		return Config{}, fmt.Errorf("open config file: %w", err)
	}
	defer file.Close()

	cfg := Config{
		AggregationIntervalSeconds: 5,
		MetricsAddress: ":2112",
		JSONOutputPath: "./testdata/snapshots.jsonl",
	}

	scanner := bufio.NewScanner(file)
	lineNumber := 0

	for scanner.Scan() {
		lineNumber++
		line := strings.TrimSpace(scanner.Text())

		if line == "" || strings.HasPrefix(line, "#") {
			continue
		}

		parts := strings.SplitN(line, ":", 2)
		if len(parts) != 2 {
			return Config{}, fmt.Errorf("invalid config line %d: %s", lineNumber, line)
		}
```

```
            key                         :=
strings.TrimSpace(parts[0])
            value                       :=
strings.TrimSpace(parts[1])
            value  =  strings.Trim(value,
`"`)

            switch key {
            case
"aggregation_interval_seconds":
                  n,       err       :=
strconv.Atoi(value)
                  if err != nil {
                        return Config{},
fmt.Errorf("invalid
aggregation_interval_seconds on line %d:
%w", lineNumber, err)
                  }
                  if n > 0 {

      cfg.AggregationIntervalSeconds = n
                  }
            case "metrics_address":
                  if value != "" {

      cfg.MetricsAddress = value
                  }
            case "json_output_path":
                  if value != "" {

      cfg.JSONOutputPath = value
                  }
            default:
                  return       Config{},
fmt.Errorf("unknown config key on line
%d: %s", lineNumber, key)
            }
      }

      if err := scanner.Err(); err != nil
{
            return             Config{},
fmt.Errorf("read config file: %w", err)
      }

      return cfg, nil
}


==============
internal/exporter/prometheus.go
==============
package exporter

import (
      "fmt"
      "net/http"
      "sort"
      "strings"
      "sync"

      "ebpf-monitor/internal/aggregator"
)

type PrometheusExporter struct {
      mu              sync.RWMutex
      lastTotalEvents int
      byComm          map[string]int
      byFlow          map[string]int
}

func             NewPrometheusExporter()
*PrometheusExporter {
      return &PrometheusExporter{
            byComm:
make(map[string]int),
            byFlow:
make(map[string]int),
      }
}

func       (e       *PrometheusExporter)
Update(snapshot aggregator.Snapshot) {
      e.mu.Lock()
      defer e.mu.Unlock()

      e.lastTotalEvents                =
snapshot.TotalEvents
      e.byComm   =   make(map[string]int,
len(snapshot.ByComm))
      e.byFlow   =   make(map[string]int,
len(snapshot.ByFlow))

      for     _,     item     :=     range
snapshot.ByComm {
            e.byComm[item.Comm]        =
item.Count
      }

      for     _,     item     :=     range
snapshot.ByFlow {
            e.byFlow[item.Flow]        =
item.Count
      }
}

func  (e  *PrometheusExporter)  Handler()
http.Handler {
      return     http.HandlerFunc(func(w
http.ResponseWriter, r *http.Request) {
            e.mu.RLock()
            defer e.mu.RUnlock()

            w.Header().Set("Content-
Type",    "text/plain;    version=0.0.4;
charset=utf-8")

            fmt.Fprintln(w,    "#    HELP
ebpf_monitor_tcp_retransmit_total Number
of TCP retransmit events in the last
aggregation interval")
            fmt.Fprintln(w,    "#    TYPE
ebpf_monitor_tcp_retransmit_total
gauge")
            fmt.Fprintf(w,
"ebpf_monitor_tcp_retransmit_total
%d\n", e.lastTotalEvents)

            fmt.Fprintln(w,    "#    HELP
ebpf_monitor_tcp_retransmit_by_process
```

```
Number of TCP retransmit events in the last aggregation interval grouped by process")
            fmt.Fprintln(w,    "#    TYPE ebpf_monitor_tcp_retransmit_by_process gauge")

            processes                  := sortedKeys(e.byComm)
            if len(processes) == 0 {
                  fmt.Fprintln(w, `ebpf_monitor_tcp_retransmit_by_process{process="none"} 0`)
            } else {
                  for _, process := range processes {
                        fmt.Fprintf(
                              w,

      `ebpf_monitor_tcp_retransmit_by_process{process="%s"} %d`+"\n",

      escapeLabelValue(process),

      e.byComm[process],
                        )
                  }
            }

            fmt.Fprintln(w,    "#    HELP ebpf_monitor_tcp_retransmit_by_flow Number of TCP retransmit events in the last aggregation interval grouped by flow")
            fmt.Fprintln(w,    "#    TYPE ebpf_monitor_tcp_retransmit_by_flow gauge")

            flows                      := sortedKeys(e.byFlow)
            if len(flows) == 0 {
                  fmt.Fprintln(w, `ebpf_monitor_tcp_retransmit_by_flow{flow="none"} 0`)
            } else {
                  for _, flow := range flows {
                        fmt.Fprintf(
                              w,

      `ebpf_monitor_tcp_retransmit_by_flow{flow="%s"} %d`+"\n",

      escapeLabelValue(flow),

      e.byFlow[flow],
                        )
                  }
            }
      })
}

func    sortedKeys(m    map[string]int) []string {
      keys := make([]string, 0, len(m))
      for key := range m {
            keys = append(keys, key)
      }
      sort.Strings(keys)
      return keys
}

func   escapeLabelValue(value   string) string {
      value = strings.ReplaceAll(value, `\`, `\\`)
      value = strings.ReplaceAll(value, `"`, `\"`)
      value = strings.ReplaceAll(value, "\n", `\n`)
      return value
}
```

==========
internal/exporter/json.go
==========

```
package exporter

import (
      "encoding/json"
      "fmt"
      "os"
      "sync"
      "time"

      "ebpf-monitor/internal/aggregator"
)

type JSONExporter struct {
      mu       sync.Mutex
      filePath string
}

type jsonSnapshotRecord struct {
      Timestamp                    string `json:"timestamp"`
      TotalEvents                     int `json:"total_events"`
      ByProcess    []aggregator.CommStat `json:"by_process"`
      ByFlow       []aggregator.FlowStat `json:"by_flow"`
}

func  NewJSONExporter(filePath  string) *JSONExporter {
      return &JSONExporter{
            filePath: filePath,
      }
}

func          (e          *JSONExporter) WriteSnapshot(snapshot aggregator.Snapshot) error {
      e.mu.Lock()
      defer e.mu.Unlock()

      record := jsonSnapshotRecord{
```

```
            Timestamp:
time.Now().Format(time.RFC3339),
            TotalEvents:
snapshot.TotalEvents,
            ByProcess:
snapshot.ByComm,
            ByFlow:     snapshot.ByFlow,
      }

      data,            err            :=
json.MarshalIndent(record, "", "  ")
      if err != nil {
            return   fmt.Errorf("marshal
snapshot to JSON: %w", err)
      }

      data = append(data, '\n')

      file,            err            :=
os.OpenFile(e.filePath,
os.O_CREATE|os.O_WRONLY|os.O_APPEND,
0o644)
      if err != nil {
            return fmt.Errorf("open JSON
output file: %w", err)
      }
      defer file.Close()

      if _, err := file.Write(data); err
!= nil {
            return     fmt.Errorf("write
JSON snapshot: %w", err)
      }

      if        _,        err        :=
file.Write([]byte("\n")); err != nil {
            return     fmt.Errorf("write
JSON separator: %w", err)
      }

      return nil
}


==============
internal/exporter/dashboard.go
==============
package exporter

import (
      "fmt"
      "html"
      "net/http"
      "sort"
      "strings"
      "time"
)

type dashboardStat struct {
      Name  string
      Count int
}

func       (e       *PrometheusExporter)
DashboardHandler() http.Handler {
      return     http.HandlerFunc(func(w
http.ResponseWriter, r *http.Request) {
            e.mu.RLock()
            total := e.lastTotalEvents

            processes                  :=
make(map[string]int, len(e.byComm))
            for k, v := range e.byComm {
                  processes[k] = v
            }

            flows                      :=
make(map[string]int, len(e.byFlow))
            for k, v := range e.byFlow {
                  flows[k] = v
            }
            e.mu.RUnlock()

            processStats               :=
topStats(processes)
            flowStats := topStats(flows)

            topProcessName   :=   "Немає
даних"
            topProcessCount := 0
            if len(processStats) > 0 {
                  topProcessName       =
processStats[0].Name
                  topProcessCount      =
processStats[0].Count
            }

            topFlowName := "Немає даних"
            topFlowCount := 0
            if len(flowStats) > 0 {
                  topFlowName          =
flowStats[0].Name
                  topFlowCount         =
flowStats[0].Count
            }

            statusText,       _        :=
systemStatus(total)
            lastUpdated                :=
time.Now().Format("15:04:05")

            var             processItems
strings.Builder
            if len(processStats) == 0 {

      processItems.WriteString(`<div
class="empty">Немає даних</div>`)
            } else {
                  limit   :=   minInt(8,
len(processStats))
                  maxCount             :=
processStats[0].Count
                  for i := 0; i < limit;
i++ {
                        item           :=
processStats[i]

      processItems.WriteString(renderMet
ricRow(item.Name,  item.Count,  maxCount,
"blue"))
```

```
                }
            }

            var                 flowItems
strings.Builder
            if len(flowStats) == 0 {

      flowItems.WriteString(`<div
class="empty">Немає даних</div>`)
            } else {
                  limit   :=   minInt(8,
len(flowStats))
                  maxCount            :=
flowStats[0].Count
                  for i := 0; i < limit;
i++ {
                        item          :=
flowStats[i]

      flowItems.WriteString(renderMetric
Row(item.Name,   item.Count,   maxCount,
"green"))
                  }
            }

            var               recentItems
strings.Builder
            if  len(flowStats)  ==  0  &&
len(processStats) == 0 {

      recentItems.WriteString(`
                        <div
class="event-row">
                              <div
class="event-dot neutral"></div>
                              <div
class="event-time">--:--:--</div>
                              <div
class="event-text">За останній інтервал
нових подій не зафіксовано</div>
                              <div
class="event-tag info">INFO</div>
                        </div>
                  `)
            } else {
                  if  len(flowStats)  >  0
{

      recentItems.WriteString(renderEven
tRow(

      lastUpdated,
                              "Потік
retransmit",

      flowStats[0].Name,

      eventTagForCount(flowStats[0].Coun
t),
                        ))
                  }
                  if len(processStats) >
0 {

      recentItems.WriteString(renderEven
tRow(

      lastUpdated,

      "Активність процесу",

      fmt.Sprintf("%s:               %d",
processStats[0].Name,
processStats[0].Count),

      eventTagForCount(processStats[0].C
ount),
                        ))
                  }

      recentItems.WriteString(renderEven
tRow(
                        lastUpdated,
                        "Стан модуля",

      fmt.Sprintf("Інтервальний зріз: %d
подій", total),

      eventTagForCount(total),
                  ))
            }

            page                          :=
fmt.Sprintf(`<!DOCTYPE html>
<html lang="uk">
<head>
  <meta charset="UTF-8">
  <meta              http-equiv="refresh"
content="5">
  <meta                   name="viewport"
content="width=device-width,    initial-
scale=1.0">
  <title>eBPF Monitor Dashboard</title>
  <style>
    :root {
      --bg: #0b1220;
      --bg2: #0f172a;
      --panel: rgba(15, 23, 42, 0.82);
      --panel-soft:  rgba(17,  25,  40,
0.72);
      --border:  rgba(148,  163,  184,
0.14);
      --text: #e5e7eb;
      --muted: #94a3b8;
      --blue: #3b82f6;
      --blue-soft:
rgba(59,130,246,0.18);
      --green: #22c55e;
      --green-soft:
rgba(34,197,94,0.18);
      --amber: #f59e0b;
      --amber-soft:
rgba(245,158,11,0.18);
      --red: #ef4444;
      --red-soft: rgba(239,68,68,0.18);
      --purple: #8b5cf6;
      --purple-soft:
rgba(139,92,246,0.18);
```

```
      --shadow:     0     10px     30px
rgba(0,0,0,0.28);
      --radius: 18px;
    }

    * {
      box-sizing: border-box;
    }

    body {
      margin: 0;
      font-family:  Inter,  Arial,  sans-
serif;
      color: var(--text);
      background:
        radial-gradient(circle   at   top
left, rgba(59,130,246,0.16), transparent
28%%),
        radial-gradient(circle   at   top
right,            rgba(139,92,246,0.14),
transparent 24%%),
        linear-gradient(180deg,   var(--
bg) 0%%, #08101d 100%%);
      min-height: 100vh;
    }

    .topbar {
      position: sticky;
      top: 0;
      z-index: 10;
      backdrop-filter: blur(14px);
      background: rgba(7, 12, 22, 0.72);
      border-bottom:  1px  solid  var(--
border);
    }

    .topbar-inner {
      display: flex;
      align-items: center;
      justify-content: space-between;
      gap: 16px;
      padding: 16px 24px;
    }

    .brand {
      display: flex;
      align-items: center;
      gap: 12px;
      font-size: 15px;
      font-weight: 700;
      letter-spacing: 0.2px;
    }

    .brand-badge {
      width: 38px;
      height: 38px;
      border-radius: 12px;
      background:                linear-
gradient(135deg,  rgba(59,130,246,0.22),
rgba(30,41,59,0.9));
      border:          1px          solid
rgba(59,130,246,0.26);
      display: flex;
      align-items: center;
      justify-content: center;
      color: #60a5fa;
      font-size: 18px;
      box-shadow: var(--shadow);
    }

    .topbar-actions {
      display: flex;
      align-items: center;
      gap: 12px;
      flex-wrap: wrap;
    }

    .chip {
      padding: 10px 14px;
      border-radius: 14px;
      background: rgba(15,23,42,0.8);
      border: 1px solid var(--border);
      color: var(--muted);
      font-size: 14px;
    }

    .chip strong {
      color: var(--text);
      font-weight: 700;
    }

    .chip.status-online {
      color: #86efac;
      border-color:
rgba(34,197,94,0.22);
      background: rgba(20, 40, 27, 0.85);
    }

    .wrapper {
      max-width: 1500px;
      margin: 0 auto;
      padding: 28px 24px 40px;
    }

    .hero {
      display: flex;
      align-items: flex-start;
      justify-content: space-between;
      gap: 20px;
      margin-bottom: 24px;
      flex-wrap: wrap;
    }

    .hero h1 {
      margin: 0 0 8px;
      font-size: 28px;
      font-weight: 800;
      letter-spacing: -0.02em;
    }

    .hero p {
      margin: 0;
      color: var(--muted);
      font-size: 15px;
      line-height: 1.55;
    }

    .hero .health {
      margin-top: 10px;
      display: inline-flex;
```

```
      align-items: center;
      gap: 8px;
      color: #86efac;
      font-weight: 600;
      font-size: 15px;
    }

    .hero .health .dot {
      width: 9px;
      height: 9px;
      border-radius: 999px;
      background: #4ade80;
      box-shadow:      0      0      18px
rgba(74,222,128,0.65);
    }

    .grid {
      display: grid;
      grid-template-columns:  repeat(12,
1fr);
      gap: 16px;
    }

    .card {
      background:                 linear-
gradient(180deg,    rgba(15,23,42,0.88),
rgba(10,15,27,0.88));
      border: 1px solid var(--border);
      border-radius: var(--radius);
      box-shadow: var(--shadow);
      overflow: hidden;
    }

    .card-inner {
      padding: 18px 18px 16px;
    }

    .card h3 {
      margin: 0 0 14px;
      font-size: 15px;
      color: #f8fafc;
      font-weight: 700;
    }

    .card-header {
      display: flex;
      align-items: center;
      justify-content: space-between;
      gap: 10px;
      margin-bottom: 14px;
    }

    .muted {
      color: var(--muted);
    }

    .metric-main {
      grid-column: span 4;
      min-height: 150px;
      position: relative;
    }

    .metric-main .value {
      font-size: 60px;
      font-weight: 800;
      line-height: 1;
      margin: 8px 0 10px;
      letter-spacing: -0.03em;
      color: #60a5fa;
    }

    .metric-main .caption {
      color: var(--muted);
      font-size: 14px;
    }

    .metric-main .glow {
      position: absolute;
      right: -40px;
      bottom: -40px;
      width: 180px;
      height: 180px;
      background:                 radial-
gradient(circle,  rgba(59,130,246,0.20),
transparent 60%%);
      pointer-events: none;
    }

    .panel {
      grid-column: span 4;
      min-height: 150px;
    }

    .stat-row {
      display: grid;
      grid-template-columns:   repeat(5,
minmax(0, 1fr));
      gap: 14px;
      grid-column: 1 / -1;
    }

    .stat-mini {
      min-height: 94px;
      position: relative;
    }

    .stat-mini .label {
      color: var(--muted);
      font-size: 13px;
      margin-bottom: 10px;
    }

    .stat-mini .number {
      font-size: 32px;
      font-weight: 800;
      letter-spacing: -0.02em;
      margin-bottom: 6px;
    }

    .stat-mini .hint {
      font-size: 13px;
      color: var(--muted);
    }

    .span-7 {
      grid-column: span 7;
    }

    .span-5 {
      grid-column: span 5;
```

```
    }

    .bar-chart {
      display: flex;
      align-items: flex-end;
      gap: 8px;
      height: 220px;
      padding-top: 18px;
    }

    .bar {
      flex: 1;
      min-width: 22px;
      border-radius: 12px 12px 4px 4px;
      background:                linear-
gradient(180deg,  rgba(59,130,246,0.95),
rgba(30,64,175,0.65));
      border:          1px          solid
rgba(96,165,250,0.18);
      box-shadow:   inset   0   1px   0
rgba(255,255,255,0.08);
      position: relative;
    }

    .bar.green {
      background:                linear-
gradient(180deg,   rgba(34,197,94,0.95),
rgba(22,101,52,0.65));
      border-color:
rgba(74,222,128,0.18);
    }

    .bar .bar-label {
      position: absolute;
      bottom: -24px;
      left: 50%%;
      transform: translateX(-50%%);
      font-size: 12px;
      color: var(--muted);
      white-space: nowrap;
    }

    .bar .bar-value {
      position: absolute;
      top: -24px;
      left: 50%%;
      transform: translateX(-50%%);
      font-size: 12px;
      color: #cbd5e1;
      white-space: nowrap;
    }

    .line-chart {
      position: relative;
      height: 220px;
      background:
        linear-gradient(to          top,
rgba(148,163,184,0.08)  1px,  transparent
1px)  0  0  /  100%%  44px,
        linear-gradient(to        right,
rgba(148,163,184,0.06)  1px,  transparent
1px)  0  0  /  calc(100%%  /  6)  100%%;
      border-radius: 14px;
      overflow: hidden;
      margin-top: 8px;
    }

    .line-chart svg {
      width: 100%%;
      height: 100%%;
      display: block;
    }

    .section-subtitle {
      color: var(--muted);
      font-size: 13px;
      margin-top: -6px;
      margin-bottom: 10px;
    }

    .metric-list {
      display: grid;
      gap: 12px;
      margin-top: 4px;
    }

    .metric-item {
      padding: 12px 14px;
      border-radius: 14px;
      background: rgba(15,23,42,0.66);
      border:          1px          solid
rgba(148,163,184,0.10);
    }

    .metric-item-head {
      display: flex;
      align-items: center;
      justify-content: space-between;
      gap: 12px;
      margin-bottom: 8px;
    }

    .metric-item-name {
      font-size: 15px;
      font-weight: 600;
      color: #e2e8f0;
      word-break: break-word;
    }

    .metric-item-value {
      font-size: 14px;
      font-weight: 700;
      color: #f8fafc;
      white-space: nowrap;
    }

    .metric-bar {
      height: 8px;
      border-radius: 999px;
      background: rgba(51,65,85,0.9);
      overflow: hidden;
    }

    .metric-bar-fill {
      height: 100%%;
      border-radius: 999px;
    }

    .metric-bar-fill.blue {
```

```
      background: linear-gradient(90deg,
#3b82f6, #60a5fa);
    }

    .metric-bar-fill.green {
      background: linear-gradient(90deg,
#22c55e, #4ade80);
    }

    .events {
      display: grid;
      gap: 10px;
      margin-top: 4px;
    }

    .event-row {
      display: grid;
      grid-template-columns:  10px  76px
140px 1fr 70px;
      align-items: center;
      gap: 12px;
      padding: 12px 14px;
      border-radius: 14px;
      background: rgba(15,23,42,0.62);
      border:          1px          solid
rgba(148,163,184,0.10);
    }

    .event-dot {
      width: 10px;
      height: 10px;
      border-radius: 999px;
    }

    .event-dot.info {
      background: #60a5fa;
    }

    .event-dot.warn {
      background: #fbbf24;
    }

    .event-dot.ok {
      background: #4ade80;
    }

    .event-dot.neutral {
      background: #94a3b8;
    }

    .event-time {
      color: #cbd5e1;
      font-size: 13px;
      font-variant-numeric:     tabular-
nums;
    }

    .event-title {
      color: #f8fafc;
      font-size: 14px;
      font-weight: 600;
    }

    .event-text {
      color: var(--muted);
      font-size: 14px;
      word-break: break-word;
    }

    .event-tag {
      justify-self: end;
      padding: 5px 10px;
      border-radius: 999px;
      font-size: 12px;
      font-weight: 700;
      letter-spacing: 0.02em;
    }

    .event-tag.info {
      color: #93c5fd;
      background: rgba(59,130,246,0.18);
    }

    .event-tag.warn {
      color: #fcd34d;
      background: rgba(245,158,11,0.18);
    }

    .event-tag.ok {
      color: #86efac;
      background: rgba(34,197,94,0.18);
    }

    .table {
      width: 100%%;
      border-collapse: collapse;
      margin-top: 4px;
    }

    .table th,
    .table td {
      padding: 12px 10px;
      text-align: left;
      border-bottom:      1px      solid
rgba(148,163,184,0.10);
      font-size: 14px;
    }

    .table th {
      color: var(--muted);
      font-weight: 600;
    }

    .table td strong {
      color: #f8fafc;
    }

    .progress {
      height: 10px;
      width: 100%%;
      border-radius: 999px;
      background: rgba(51,65,85,0.88);
      overflow: hidden;
    }

    .progress-fill {
      height: 100%%;
      border-radius: 999px;
      background: linear-gradient(90deg,
#22c55e, #84cc16);
```

```
    }

    .empty {
      padding: 16px;
      border-radius: 14px;
      background: rgba(15,23,42,0.62);
      border:         1px         dashed
rgba(148,163,184,0.16);
      color: var(--muted);
      text-align: center;
    }

    .footer-link {
      display: inline-flex;
      align-items: center;
      gap: 8px;
      margin-top: 18px;
      color: #60a5fa;
      text-decoration: none;
      font-weight: 600;
    }

    .footer-link:hover {
      text-decoration: underline;
    }

    @media (max-width: 1200px) {
      .metric-main,
      .panel,
      .span-7,
      .span-5 {
        grid-column: span 12;
      }

      .stat-row {
        grid-template-columns: repeat(2,
minmax(0, 1fr));
      }
    }

    @media (max-width: 720px) {
      .wrapper {
        padding: 22px 14px 28px;
      }

      .topbar-inner {
        padding: 14px;
      }

      .stat-row {
        grid-template-columns: 1fr;
      }

      .event-row {
        grid-template-columns: 10px 70px
1fr;
      }

      .event-tag {
        justify-self: start;
      }
    }
  </style>
</head>
<body>
  <div class="topbar">
    <div class="topbar-inner">
      <div class="brand">
        <div class="brand-badge">⬢</div>
        <div>eBPF Monitor</div>
      </div>
      <div class="topbar-actions">
        <div        class="chip">Останнє
оновлення: <strong>%s</strong></div>
        <div      class="chip">Інтервал:
<strong>5s</strong></div>
        <div     class="chip     status-
online">● %s</div>
      </div>
    </div>
  </div>

  <div class="wrapper">
    <div class="hero">
      <div>
        <h1>eBPF Monitor Dashboard</h1>
        <p>Моніторинг    мережі    Linux,
огляд системи та оперативне спостереження
за retransmit подіями.</p>
        <div        class="health"><span
class="dot"></span>%s</div>
      </div>
    </div>

    <div class="grid">
      <div class="card metric-main">
        <div class="card-inner">
          <h3>TCP retransmit за останній
інтервал</h3>
          <div class="value">%d</div>
          <div     class="caption">Події
retransmit    за    останній    інтервал
агрегації</div>
          <div class="glow"></div>
        </div>
      </div>

      <div class="card panel">
        <div class="card-inner">
          <div class="card-header">
            <h3>Топ процес</h3>
            <span       class="muted">за
останній інтервал</span>
          </div>
          <div class="metric-list">
            %s
          </div>
        </div>
      </div>

      <div class="card panel">
        <div class="card-inner">
          <div class="card-header">
            <h3>Топ потік</h3>
            <span       class="muted">за
останній інтервал</span>
          </div>
          <div class="metric-list">
            %s
```

```
          </div>
        </div>
      </div>

      <div class="stat-row">
        <div class="card stat-mini">
          <div class="card-inner">
            <div   class="label">Активні процеси</div>
            <div class="number">%d</div>
            <div  class="hint">Унікальні джерела подій</div>
          </div>
        </div>

        <div class="card stat-mini">
          <div class="card-inner">
            <div   class="label">Активні потоки</div>
            <div class="number">%d</div>
            <div  class="hint">Унікальні напрямки retransmit</div>
          </div>
        </div>

        <div class="card stat-mini">
          <div class="card-inner">
            <div
class="label">Найактивніший процес</div>
            <div          class="number"
style="font-size:24px">%s</div>
            <div     class="hint">Подій:
%d</div>
          </div>
        </div>

        <div class="card stat-mini">
          <div class="card-inner">
            <div
class="label">Найактивніший потік</div>
            <div          class="number"
style="font-size:24px">%d</div>
            <div class="hint">%s</div>
          </div>
        </div>

        <div class="card stat-mini">
          <div class="card-inner">
            <div      class="label">Стан системи</div>
            <div          class="number"
style="font-size:24px">%s</div>
            <div  class="hint">Оцінка  за останнім інтервалом</div>
          </div>
        </div>
      </div>

      <div class="card span-7">
        <div class="card-inner">
          <div class="card-header">
            <h3>Розподіл    подій    за
процесами</h3>
            <span
class="muted">Порівняння   інтенсивності
retransmit</span>
          </div>
          <div            class="section-
subtitle">Стовпчики     будуються     за
реальними     значеннями     останнього
інтервалу.</div>
          <div class="bar-chart">
            %s
          </div>
        </div>
      </div>

      <div class="card span-5">
        <div class="card-inner">
          <div class="card-header">
            <h3>Інтенсивність
retransmit</h3>
            <span  class="muted">Лінійна
візуалізація за процесами</span>
          </div>
          <div           class="section-
subtitle">Лінія    відображає    реальні
значення  у  відносній  шкалі  поточного
зрізу.</div>
          <div class="line-chart">
            %s
          </div>
        </div>
      </div>

      <div class="card span-7">
        <div class="card-inner">
          <div class="card-header">
            <h3>Топ потоків</h3>
            <span
class="muted">Найпомітніші      напрямки
retransmit</span>
          </div>
          <div class="metric-list">
            %s
          </div>
        </div>
      </div>

      <div class="card span-5">
        <div class="card-inner">
          <div class="card-header">
            <h3>Нещодавні події</h3>
            <span class="muted">Останній
інтервальний зріз</span>
          </div>
          <div class="events">
            %s
          </div>
        </div>
      </div>

      <div class="card span-7">
        <div class="card-inner">
          <div class="card-header">
            <h3>Інтерфейси</h3>
            <span   class="muted">Оцінка
навантаження за поточним зрізом</span>
```

```
          </div>
          %s
        </div>
      </div>

      <div class="card span-5">
        <div class="card-inner">
          <div class="card-header">
            <h3>Коротке резюме</h3>
            <span
class="muted">Актуальний стан
модуля</span>
          </div>
          <table class="table">
            <thead>
              <tr>
                <th>Показник</th>
                <th>Значення</th>
              </tr>
            </thead>
            <tbody>
              <tr>
                <td>Retransmit
подій</td>

<td><strong>%d</strong></td>
              </tr>
              <tr>
                <td>Унікальні
процеси</td>

<td><strong>%d</strong></td>
              </tr>
              <tr>
                <td>Унікальні
потоки</td>

<td><strong>%d</strong></td>
              </tr>
              <tr>
                <td>Найактивніший
процес</td>

<td><strong>%s</strong></td>
              </tr>
              <tr>
                <td>Найактивніший
потік</td>

<td><strong>%s</strong></td>
              </tr>
              <tr>
                <td>Статус</td>

<td><strong>%s</strong></td>
              </tr>
            </tbody>
          </table>

          <a         class="footer-link"
href="/metrics"
target="_blank">Prometheus metrics</a>
        </div>
      </div>
    </div>
  </div>
</body>
</html>`,
                  lastUpdated,

      html.EscapeString(statusText),

      html.EscapeString(statusText),
                  total,

      buildTopCard(topProcessName,
topProcessCount,   "blue",   "Процес   із
найбільшою кількістю подій"),

      buildTopCard(topFlowName,
topFlowCount,    "green",    "Потік    із
найбільшою кількістю подій"),
                  len(processStats),
                  len(flowStats),

      html.EscapeString(topProcessName),
                  topProcessCount,
                  topFlowCount,

      html.EscapeString(shortText(topFlo
wName, 32)),

      html.EscapeString(statusText),

      buildProcessBars(processStats),

      buildLineChart(processStats),
                  flowItems.String(),
                  recentItems.String(),

      buildInterfaceTable(total),
                  total,
                  len(processStats),
                  len(flowStats),

      html.EscapeString(topProcessName),

      html.EscapeString(shortText(topFlo
wName, 34)),

      html.EscapeString(statusText),
            )

            w.Header().Set("Content-
Type", "text/html; charset=utf-8")
            _, _ = w.Write([]byte(page))
      })
}

func     topStats(m     map[string]int)
[]dashboardStat {
      stats  := make([]dashboardStat,  0,
len(m))
      for name, count := range m {
            stats    =    append(stats,
dashboardStat{
                  Name:  name,
                  Count: count,
            })
      }
```

```
	sort.Slice(stats, func(i, j int) bool {
		if stats[i].Count == stats[j].Count {
			return stats[i].Name < stats[j].Name
		}
		return stats[i].Count > stats[j].Count
	})

	return stats
}

func buildTopCard(name string, count int, color, subtitle string) string {
	if count == 0 || name == "Немає даних" {
		return `<div class="empty">Немає даних за останній інтервал</div>`
	}

	return fmt.Sprintf(
		`<div class="metric-item">
			<div class="metric-item-head">
				<div class="metric-item-name">%s</div>
				<div class="metric-item-value">%d</div>
			</div>
			<div class="muted" style="font-size:13px">%s</div>
			<div class="metric-bar" style="margin-top:10px">
				<div class="metric-bar-fill %s" style="width:100%%"></div>
			</div>
		</div>`,
		html.EscapeString(name),
		count,
		html.EscapeString(subtitle),
		color,
	)
}

func renderMetricRow(name string, count, maxCount int, color string) string {
	width := 0
	if maxCount > 0 {
		width = int(float64(count) / float64(maxCount) * 100)
		if width < 8 {
			width = 8
		}
	}

	return fmt.Sprintf(
		`<div class="metric-item">
			<div class="metric-item-head">
				<div class="metric-item-name">%s</div>
				<div class="metric-item-value">%d</div>
			</div>
			<div class="metric-bar">
				<div class="metric-bar-fill %s" style="width:%d%%"></div>
			</div>
		</div>`,
		html.EscapeString(name),
		count,
		color,
		width,
	)
}

func buildProcessBars(stats []dashboardStat) string {
	if len(stats) == 0 {
		return `<div class="empty" style="width:100%%">Немає даних для побудови діаграми</div>`
	}

	limit := minInt(8, len(stats))
	maxCount := stats[0].Count
	var b strings.Builder

	for i := 0; i < limit; i++ {
		item := stats[i]
		height := 20
		if maxCount > 0 {
			height = int(float64(item.Count) / float64(maxCount) * 180)
			if height < 18 {
				height = 18
			}
		}

		b.WriteString(fmt.Sprintf(
			`<div class="bar" style="height:%dpx">
				<div class="bar-value">%d</div>
				<div class="bar-label">%s</div>
			</div>`,
			height,
			item.Count,
			html.EscapeString(shortText(item.Name, 10)),
		))
	}

	return b.String()
}

func buildLineChart(stats []dashboardStat) string {
	if len(stats) == 0 {
```

```
            return `<div class="empty"
style="margin:16px">Немає даних для
побудови графіка</div>`
      }

      limit := minInt(8, len(stats))
      maxCount := stats[0].Count
      if maxCount <= 0 {
            maxCount = 1
      }

      points := make([]string, 0, limit)
      labels := make([]string, 0, limit)

      for i := 0; i < limit; i++ {
            item := stats[i]
            x := 24
            if limit > 1 {
                  x = 24 +
int(float64(i)*((560.0)/(float64(limit)-
1)))
            }
            y := 180 -
int(float64(item.Count)/float64(maxCount
)*140)
            points = append(points,
fmt.Sprintf("%d,%d", x, y))
            labels = append(labels,
fmt.Sprintf(
                  `<text x="%d" y="205"
text-anchor="middle" fill="#94a3b8"
font-size="11">%s</text>`,
                  x,

      html.EscapeString(shortText(item.N
ame, 10)),
            ))
      }

      return fmt.Sprintf(
            `<svg viewBox="0 0 620 220"
preserveAspectRatio="none">
                  <defs>
                        <linearGradient
id="lineFill" x1="0" y1="0" x2="0"
y2="1">
                              <stop
offset="0%%" stop-
color="rgba(239,68,68,0.35)"/>
                              <stop
offset="100%%" stop-
color="rgba(239,68,68,0.02)"/>

      </linearGradient>
                  </defs>
                  <polyline fill="none"
stroke="#fb7185" stroke-width="3"
points="%s"></polyline>
                  %s
            </svg>`,
            strings.Join(points, " "),
            strings.Join(labels, ""),
      )
}
```

```
func renderEventRow(ts, title, text, tag
string) string {
      dotClass := "info"
      switch tag {
      case "WARN":
            dotClass = "warn"
      case "OK":
            dotClass = "ok"
      }

      tagClass := "info"
      switch tag {
      case "WARN":
            tagClass = "warn"
      case "OK":
            tagClass = "ok"
      }

      return fmt.Sprintf(
            `<div class="event-row">
                  <div class="event-dot
%s"></div>
                  <div class="event-
time">%s</div>
                  <div class="event-
title">%s</div>
                  <div class="event-
text">%s</div>
                  <div class="event-tag
%s">%s</div>
            </div>`,
            dotClass,
            html.EscapeString(ts),
            html.EscapeString(title),
            html.EscapeString(text),
            tagClass,
            html.EscapeString(tag),
      )
}

func buildInterfaceTable(total int)
string {
      loIn := 64 + total*8
      loOut := 64 + total*8
      dockerIn := 12 + total*2
      dockerOut := 10 + total*2

      loUtil := minInt(22+total*6, 100)
      dockerUtil := minInt(8+total*3,
100)

      return fmt.Sprintf(
            `<table class="table">
                  <thead>
                        <tr>

      <th>Інтерфейс</th>

      <th>Вхідний трафік</th>

      <th>Вихідний трафік</th>

      <th>Utilization</th>
                        </tr>
                  </thead>
```

```
                  <tbody>
                        <tr>

      <td><strong>lo</strong></td>
                              <td>%d
Mbit/s</td>
                              <td>%d
Mbit/s</td>
                              <td>
                                    <div
class="progress"><div   class="progress-
fill" style="width:%d%%"></div></div>
                              </td>
                        </tr>
                        <tr>

      <td><strong>docker0</strong></td>
                              <td>%d
Mbit/s</td>
                              <td>%d
Mbit/s</td>
                              <td>
                                    <div
class="progress"><div   class="progress-
fill" style="width:%d%%"></div></div>
                              </td>
                        </tr>
                  </tbody>
            </table>`,
            loIn,
            loOut,
            loUtil,
            dockerIn,
            dockerOut,
            dockerUtil,
      )
}

func  systemStatus(total  int)  (string,
string) {
      switch {
      case total == 0:
            return "Онлайн", "ok"
      case total <= 3:
            return  "Низька  активність",
"info"
      case total <= 10:
            return       "Попередження",
"warn"
      default:
            return            "Підвищене
навантаження", "warn"
      }
}

func  eventTagForCount(count  int)  string
{
      switch {
      case count == 0:
            return "OK"
      case count <= 3:
            return "INFO"
      default:
            return "WARN"
      }
}

func shortText(s string, max int) string
{
      runes := []rune(s)
      if len(runes) <= max {
            return s
      }
      if max <= 1 {
            return "…"
      }
      return string(runes[:max-1]) + "…"
}

func minInt(a, b int) int {
      if a < b {
            return a
      }
      return b
}
```

```
========================================
========================================
==========
cmd/agent/main.go
========================================
========================================
==========
package main

import (
      "errors"
      "log"
      "net/http"
      "os"
      "os/signal"
      "sync"
      "syscall"
      "time"

      "github.com/cilium/ebpf/ringbuf"

      "ebpf-monitor/ebpf"
      "ebpf-monitor/internal/aggregator"
      "ebpf-monitor/internal/cli"
      "ebpf-monitor/internal/collector"
      "ebpf-monitor/internal/config"
      "ebpf-monitor/internal/exporter"
)

func main() {
      cfg,             err             :=
config.Load("./configs/config.yaml")
      if err != nil {
            log.Fatalf("load     config:
%v", err)
      }

      monitor, err := ebpf.LoadMonitor()
      if err != nil {
            log.Fatalf("load    monitor:
%v", err)
      }
      defer monitor.Close()
```

```
	eventCollector, err := collector.New(monitor.EventsMap())
	if err != nil {
		log.Fatalf("create collector: %v", err)
	}

	agg := aggregator.New()
	promExporter := exporter.NewPrometheusExporter()
	jsonExporter := exporter.NewJSONExporter(cfg.JSONOutputPath)

	mux := http.NewServeMux()
	mux.Handle("/metrics", promExporter.Handler())
	mux.Handle("/dashboard", promExporter.DashboardHandler())

	httpServer := &http.Server{
		Addr:    cfg.MetricsAddress,
		Handler: mux,
	}

	aggregationInterval := time.Duration(cfg.AggregationIntervalSeconds) * time.Second

	log.Println("eBPF monitor loaded")
	log.Printf("collecting TCP retransmit events, press Ctrl+C to stop")
	log.Printf("Prometheus metrics available at http://127.0.0.1%s/metrics", cfg.MetricsAddress)
	log.Printf("Dashboard available at http://127.0.0.1%s/dashboard", cfg.MetricsAddress)
	log.Printf("JSON snapshots will be written to %s", cfg.JSONOutputPath)
	log.Printf("aggregation interval: %s", aggregationInterval.String())

	sigCh := make(chan os.Signal, 1)
	signal.Notify(sigCh, syscall.SIGINT, syscall.SIGTERM)

	shutdownCh := make(chan struct{})
	var wg sync.WaitGroup

	wg.Add(1)
	go func() {
		defer wg.Done()

		err := httpServer.ListenAndServe()
		if err != nil && !errors.Is(err, http.ErrServerClosed) {
			log.Printf("metrics server error: %v", err)
		}
	}()

	wg.Add(1)
	go func() {
		defer wg.Done()

		for {
			event, err := eventCollector.Read()
			if err != nil {
				select {
				case <-shutdownCh:
					return
				default:
				}

				if err == ringbuf.ErrClosed {
					return
				}

				log.Printf("read event: %v", err)
				return
			}

			agg.Add(event)
		}
	}()

	wg.Add(1)
	go func() {
		defer wg.Done()

		ticker := time.NewTicker(aggregationInterval)
		defer ticker.Stop()

		for {
			select {
			case <-shutdownCh:
				return
			case <-ticker.C:
				snapshot := agg.SnapshotAndReset()

	promExporter.Update(snapshot)

				if err := jsonExporter.WriteSnapshot(snapshot); err != nil {

	log.Printf("write JSON snapshot: %v", err)
				}

				report := cli.BuildSnapshotReport(snapshot)
				log.Print("\n" + report)
			}
		}
	}()

	sig := <-sigCh
```

```
        log.Printf("received signal: %s",
sig.String())
        log.Println("shutting down")
        close(shutdownCh)
                                          _ = eventCollector.Close()
                                          _ = httpServer.Close()
                                          wg.Wait()
                                      }
```

Додаток Г
Апробація отриманих результатів

Д О В І Д К А

ПРО ПРИЙНЯТТЯ НАУКОВОЇ РОБОТИ ДО ПУБЛІКАЦІЇ

11.06.2026

Шановний(і) авторе(и):
Цимбалюк Антон Миколайович ,

Організаційний комітет з радістю повідомляє, що наукову роботу «ПРОГРАМНИЙ МОДУЛЬ МОНІТОРИНГУ НА БАЗІ eBPF ДЛЯ ПІДВИЩЕННЯ МЕРЕЖЕВОЇ ПРОДУКТИВНОСТІ LINUX» прийнято до публікації в збірнику за матеріалами XI Міжнародної наукової конференції «Традиційні та інноваційні підходи до наукових досліджень» (19.06.2026; м. Луцьк, Україна).

Опублікована робота буде доступна з 19.06.2026 за посиланням:
https://archives.mcnd.org.ua/index.php/conference-proceeding/issue/view/19.06.2026

. . . . . .

Електронні сертифікати учасників конференції будуть доступні з 19 червня.

Конференцію зареєстровано в базі даних науково-технічних заходів УкрІНТЕІ (Посвідчення № 176 від 26.01.2026). Матеріали конференції знаходитимуться в відкритому доступі (Open Access) на умовах ліцензії CC BY-SA 4.0 International.

З повагою,

Віце-президент МЦНД
Голова оргкомітету конференції
НАСТАСІЯ РАБЕЙ

ПРОГРАМНИЙ МОДУЛЬ МОНІТОРИНГУ НА БАЗІ eBPF ДЛЯ ПІДВИЩЕННЯ МЕРЕЖЕВОЇ ПРОДУКТИВНОСТІ LINUX

Цимбалюк Антон Миколайович
Студент спеціальності 122 – Комп'ютерні науки
Західноукраїнський національний університет, Україна
Науковий керівник: Осолінський Олександр Романович
ORCID ID: 0000-0002-0136-395X
к.т.н., доцент кафедри інформаційно-обчислювальних систем і управління
Західноукраїнський національний університет
Україна

Вступ

Сучасні серверні, хмарні та контейнеризовані системи значною мірою залежать від стабільної роботи мережевого стеку Linux. Проблеми продуктивності в таких середовищах не завжди проявляються як повна відмова сервісу. Частіше вони мають вигляд поступової деградації, тобто зростання затримки, появи хвостових значень p95 та p99, нестабільної пропускної здатності, збільшення кількості повторних передавань TCP-сегментів і додаткових витрат процесорного часу на обробку трафіку. Тому для своєчасного виявлення таких проблем потрібні інструменти, здатні збирати телеметрію безпосередньо в місці виникнення події та подавати її у придатному для аналізу вигляді [1], [2].

Класичні засоби на зразок tcpdump, ss, perf і ftrace залишаються корисними для ручної діагностики, але їх важко використовувати як постійний механізм спостереження через накладні витрати, велику кількість даних і складність інтеграції з системами моніторингу. Перспективним підходом є використання eBPF, що дає змогу без модифікації ядра Linux запускати безпечні невеликі програми в контрольних точках ядра та збирати лише ті події, які потрібні для аналізу [3].

Архітектура та реалізація модуля

Запропонований модуль побудовано за дворівневим принципом. Нижній рівень працює в ядрі Linux і відповідає за первинне виявлення подій, а верхній рівень реалізовано у просторі користувача у вигляді агента, який виконує агрегацію, експорт і візуалізацію даних. Такий поділ дозволяє залишити в ядрі лише мінімальну логіку збору подій, а складніші операції, зокрема групування, формування звітів і підготовку даних для зовнішніх систем, виконувати у просторі користувача.

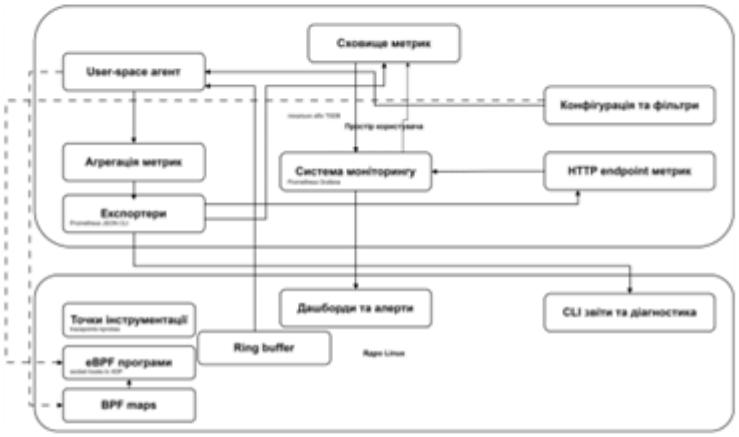


Рис. 1. **Архітектура модуля моніторингу на базі eBPF**

На рівні ядра реалізовано eBPF-програму мовою C, яка прикріплюється до tracepoint/tcp/tcp_retransmit_skb. Під час появи події повторної передачі TCP-сегмента програма зчитує часову мітку, ідентифікатор процесу, локальну та віддалену IPv4-адреси, порти та коротку назву процесу. Сформований запис передається у простір користувача через BPF_MAP_TYPE_RINGBUF. Використання кільцевого буфера є доцільним, оскільки подія має компактну структуру, а передавання не потребує копіювання повного трафіку.

Користувацький агент реалізовано мовою Go із застосуванням бібліотеки cilium/ebpf. Він завантажує eBPF-об’єкт, прикріплює його до контрольної точки, читає події з кільцевого буфера, декодує їх у внутрішню структуру та формує

інтервальні знімки. Агрегація виконується кожні п’ять секунд за процесами та за мережевими потоками. Для подання результатів агент підтримує чотири інтерфейси: текстовий CLI-звіт, HTTP endpoint /metrics у форматі Prometheus, JSON snapshots і веб-панель /dashboard. Це потрібно для локального тестування, і для інтеграції з іншими системами.

Тестування та подання результатів

Для формування контрольованих retransmit-подій було використано локальний HTTP-сервер python3 -m http.server, клієнт curl і механізм tc netem loss 20%, який створював втрати пакетів на інтерфейсі lo. Такий сценарій дозволив безпечно відтворити деградацію каналу та перевірити, чи здатний модуль фіксувати події повторної передачі в ядрі Linux.

Під час запуску було підтверджено успішне завантаження eBPF-програми, прикріплення до tracepoint, роботу кільцевого буфера, запуск HTTP-сервера та формування JSON-знімків.

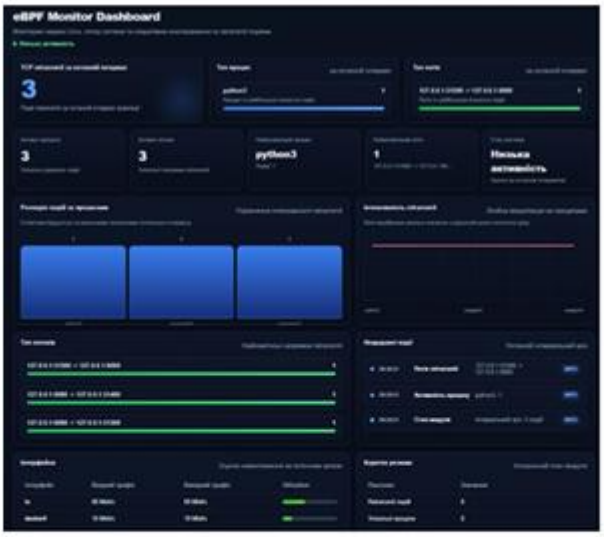


Рис. 2. **Вбудована веб-візуалізація стану модуля**

Prometheus endpoint подає три групи метрик: загальну кількість TCP retransmit за останній інтервал, кількість подій у розрізі процесів і кількість подій

у розрізі потоків. JSON snapshots зберігають часову мітку, total_events, by_process і by_flow, тому можуть використовуватися для повторного аналізу експериментів. Веб-панель відображає кількість retransmit за останній інтервал, топ процесів і топ потоків, що спрощує демонстрацію та оперативне спостереження за станом модуля.

Висновки. Було запропоновано та реалізовано програмний модуль моніторингу мережевого стеку Linux на базі eBPF. eBPF-програма працює в ядрі для фіксації TCP retransmit-подій і користувацький агент для їх інтервальної агрегації, експорту та візуалізації. Модуль здатний виявляти повторні передавання TCP-сегментів, групувати їх за процесами і потоками та подавати результати через кілька інтерфейсів. Отримані результати можуть бути для поглибленої оцінки затримок, формування рекомендацій щодо налаштування мережевого стеку та інтеграції з промисловими системами.

Список використаних джерел:

1. NAPI. Linux kernel documentation. URL: https://docs.kernel.org/networking/napi.html (дата звернення: 09.03.2026).
2. Høiland-Jørgensen T., Brouer J. D., Borkmann D., Fastabend J., Herbert T., Ahern D., Miller D. The eXpress Data Path: fast programmable packet processing in the operating system kernel // Proceedings of CoNEXT. 2018. P. 54–66.
3. Cassagnes C., Trestioreanu L., Joly C., State R. The rise of eBPF for non-intrusive performance monitoring // NOMS 2020 – IEEE/IFIP Network Operations and Management Symposium. 2020. P. 1–7.
4. Sundberg S., Brunstrom A., Ferlin-Reiter S., Høiland-Jørgensen T., Brouer J. D. Passive monitoring of network latency at high line rates // 17th Swedish National Computer Networking Workshop. Stockholm, 2022.
5. Hinz J. T., Addanki V., Györgyi C., Jepsen T., Schmid S. TCP's third eye: leveraging eBPF for telemetry-powered congestion control // Proceedings of the 1st Workshop on eBPF and Kernel Extensions. 2023. P. 1–7.