Network Telemetry Aggregation System

Author: Yaniv Bar-Lev.
Date: December 2nd, 2025
Applying for Senior Software Engineer - Nvidia.

Requirements

Functional

1. **Ingestion**:
   Simulate a feed of telemetry data (e.g., a fake data generator that updates telemetry every 10 seconds for a set of switches). These should simulate real - world metrics (e.g., random bandwidth values, latency spikes, or errors).

2. **Data Storage**: Store telemetry metrics for each switch. Keys can represent switch/server IDs, with fields representing individual metrics.

3. **Counters**:
   Telemetry data should be exposed in a REST server implementing GET http://127.0.0.1:9001/counters , which will return a CSV (matrix-based) with all the telemetry data (per switch/server, with counters' names as the columns).

4. **Metrics Server**:
   Implement two REST (GetMetric, ListMetrics) endpoints on http://127.0.0.1:8080/telemetry/.
   - `GetMetric`: Fetch the current value of a specific metric for a specific switch.
   - `ListMetrics`: Fetch a list of metric values (e.g., bandwidth consumption, latency) for all switches.

Non-Functional

1. **Performance**:

- Telemetry metrics should be available for fast API responses.
- Unexpected simultaneous queries should not degrade performance (handling a request should not be blocking)

2. **Data freshness:**
   Application should ensure data freshness with an efficient update mechanism.

3. **Observability:**
   Expose application performance/responsiveness metric collection.

4. **Logging:**
   Include logging for debugability.

Core Entities

Project structure

ufm_telemetry-aggregator

- ***ufm_aggregator_app*:** contains the telemetry application modules.

- ***ufm_telemetry_generator***: contains the telemetry generator application modules. This application includes a ***main*** module and ***readme.md*** of its own. Its main purpose is to simulate data generation for ingestion by the aggregation app.
  This application is self contained and is not part of the overall packages of the project (no ***__init__*** file)

- ***main***: the execution entry module running the aggregator application.

- ***readme.md***

- ***logs***: folder for loggings.

- ***__init__***: for defining the main parent package of the project.

Modules

- ***ufm_aggregator_app:***

- ***ufm_aggregator_web_app***: encapsulates FastApi based application endpoints and creation with ***asynccontextmanager*** telemetry lifespan to manage telemetry application resources startup/teardown.

- ***ufm_aggrtegator_server***: encapsulates data ingestion for metrics received by pulling http://127.0.0.1:9001/counters

- ***ufm_common***: contains common definitions/constants (mostly naming to avoid typos) used by the telemetry aggregator application modules. E.g. performance counters, counters types (bandwidth_usage/latency/packet_error), etc.

- ***ufm_counters_csv_parser***: contains a simple csv string parser which is used for the ingest process

- ***ufm_performance_counters***: performance counters data definitions which includes:
  - Ingest performance (total count 24h/total failures 24h and min/max/avg duration)
  - Api performance (total request 24h/total failures 24h and min/max/avg latency)

- ***ufm_performance_store***: encapsulate the performance statistics service - serves the update logic and reporting single source.

- ***ufm_telemetry_store***: encapsulate the telemetry data storage service - serves both the ingestion update process (replacing the data to latest each ingest) and reporting single source.

- ***__init__***: to define directory as a sub package of the project.

- ***ufm_aggregator_app:***
  - ***ufm_metric_generator***: encapsulates random metric generation. Single responsibility to create telemetry data for given switches based on previous data randomized.

  - ***ufm_telemetry_generator_app***: encapsulates FastApi based application endpoints and creation with ***asynccontextmanager*** counters lifespan to manage counters application resources startup/teardown.

- ***ufm_telemetry_generator_common***: contains common naming definitions/constants used by the counters generator application modules. E.g. counters types (bandwidth_usage/latency/packet_error), etc.

- ***ufm_telemetry_generator***: main telemetry counters update loop. It creates new data utilizing the *ufm_metric_generator* saving it each update to a separate alternating csv-strings for seamless service request while updating (locking the current csv for alternating after the new csv string is ready to avoid read/write collisions).

- ***ufm_telemetry_metric***: contains a class defining the full telemetry metric containing: *switch_name*, *timestamp*, *bandwidth_usage*, *latency*, *packet_error*

APIs

Endpoints

Telemetry

Running on ***localhost*** (127.0.01) on port 8000, contains the following endpoint:

- ***/telemetry/*** - serving the following REST APIs:
  - **GET *****/telemetry/get_metric***
  - **GET *****/telemetry/list_metrics***
  - ***GET /telemetry/performance***

Counters

Running on ***localhost*** (127.0.01) on port 9001, contains the following endpoint:

- ***/counters/*** - serving the following REST APIs:
  - **GET *****/counters/***

Definitions

GetMetric

GET /telemetry/get_metric?switch_name=<switch_name>&metric=<metric> → {“switch_name”: <switch_name>, “metric”: <metric_name>, “value”: <number>}

Result example:

- *{"switch_name":"sw-05","metric":"latency","value":"2572.537"}*
- *{"switch_name":"sw-03","metric":"packet_error","value":"7"}*
- *{"switch_name":"sw-02","metric":"bandwidth_usage","value":"1607.96"}*

ListMetrics

GET /telemetry/list_metrics → {"metrics":[{"server_name":<switch_name>,"bandwidth_usage":<real_value>,"latency":<real_value>,"packet_error":<int_value>,"timestamp":<utc_string_iso-format>}]}

Result example:

{"metrics":[{"server_name":"sw-01","bandwidth_usage":"358.05","latency":"1909.074","packet_error":"10","timestamp":"2025-12-01T23:08:04.855019"},{"server_name":"sw-02","bandwidth_usage":"1607.96","latency":"2452.467","packet_error":"13","timestamp":"2025-12-01T23:08:04.855063"},{"server_name":"sw-03","bandwidth_usage":"1652.89","latency":"2092.42","packet_error":"7","timestamp":"2025-12-01T23:08:04.855077"},{"server_name":"sw-04","bandwidth_usage":"608.02","latency":"2269.627","packet_error":"8","timestamp":"2025-12-01T23:08:04.855088"},{"server_name":"sw-05","bandwidth_usage":"14.88","latency":"2572.537","packet_error":"6","timestamp":"2025-12-01T23:08:04.855097"},{"server_name":"sw-06","bandwidth_usage":"1454.05","latency":"2356.426","packet_error":"10","timestamp":"2025-12-01T23:08:04.855106"},{"server_name":"sw-07","bandwidth_usage":"881.07","latency":"1320.11","packet_error":"11","timestamp":"2025-12-01T23:08:04.855115"},{"server_name":"sw-08","bandwidth_usage":"725.87","latency":"3311.024","packet_error":"13","timestamp":"2025-12-01T23:08:04.855124"},{"server_name":"sw-09","bandwidth_usage":"852.08","latency":"2413.137","packet_error":"8","timestamp":"2025-12-01T23:08:04.855132"},{"server_name":"sw-10","bandwidth_usage":"1106.09","latency":"1943.945","packet_error":"12","timestamp":"2025-12-01T23:08:04.855141"}]}

Performance

GET /telemetry/performance →
{"measure-timestamp":<utc_string_iso-format>,"total_ingest_count_24":<int_value>,"total_ingest_failures_24":<int_value>,"max_ingest_duration_ms":<real_value>,"avg_ingest_duration_ms":<real_value>,"min_ingest_duration_ms":<real_value>,"total_api_requests_24":<int_value>,
"total_api_requests_failures_24":<int_value>,"min_api_latency_ms":<real_value>,"avg_api_latency_ms":<real_value>,"max_api_latency_ms":<real_value>}

Result example:

"{"measure-timestamp": "2025-12-01T23:08:01.027441", "total_ingest_count_24": 2, "total_ingest_failures_24": 0, "max_ingest_duration_ms": 0.49432600010186434, "avg_ingest_duration_ms": 0.46324275014922023, "min_ingest_duration_ms": 0.0, "total_api_requests_24": 7, "total_api_requests_failures_24": 0,"min_api_latency_ms": 0.0, "avg_api_latency_ms": 0.5010715685784817, "max_api_latency_ms": 1.2215999886393547}"

Counters

Data Flow

The following lists highlights the data flow process:

1. Generating telemetry counters data:
    a. Generating random ***bandwidth_usage***, ***latency***, and ***packet_error*** values for a set of predefined switches and the current timestamp.
    b. Replacing previous values with new.
    c. Building a csv (a head) and saving it in an alternating csv like string.
    d. Sleeping for 10 seconds
    e. Repeat until graceful/hard stop.

2. Fetching telemetry data:
    a. With http client with 5 seconds timeout - request new telemetry counters from REST endpoint: /counters/ on localhost, port:9001
    b. Unpack csv string
    c. Replace data in telemetry store in bulk with global lock
    d. Sleeping for 10 seconds
    e. Repeat until graceful/hard stop.

3. Exposing telemetry data:
    a. Get metric:
        i. Receive request on endpoint method

    ii. Extract switch_name and metric_type
    iii. Access data store to get specific metric with no locking/blocking issue.
    iv. Reply as json

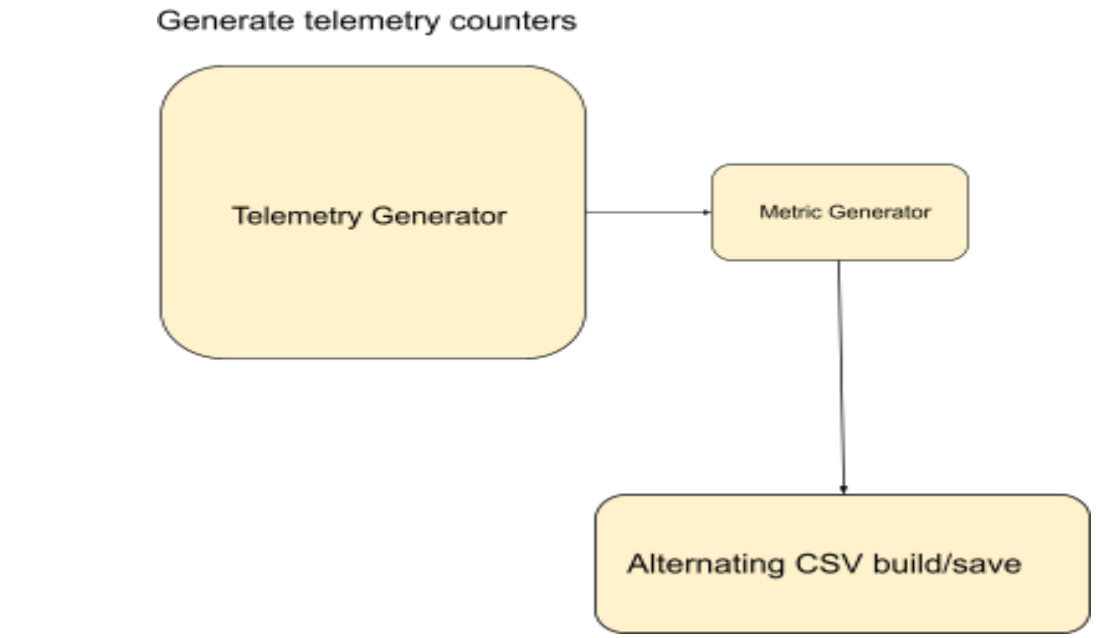
b. List metrics:
    i. Receive request on endpoint method
    ii. Access data store to get all metrics (for all servers) with no locking/blocking issue.
    iii. Reply as json

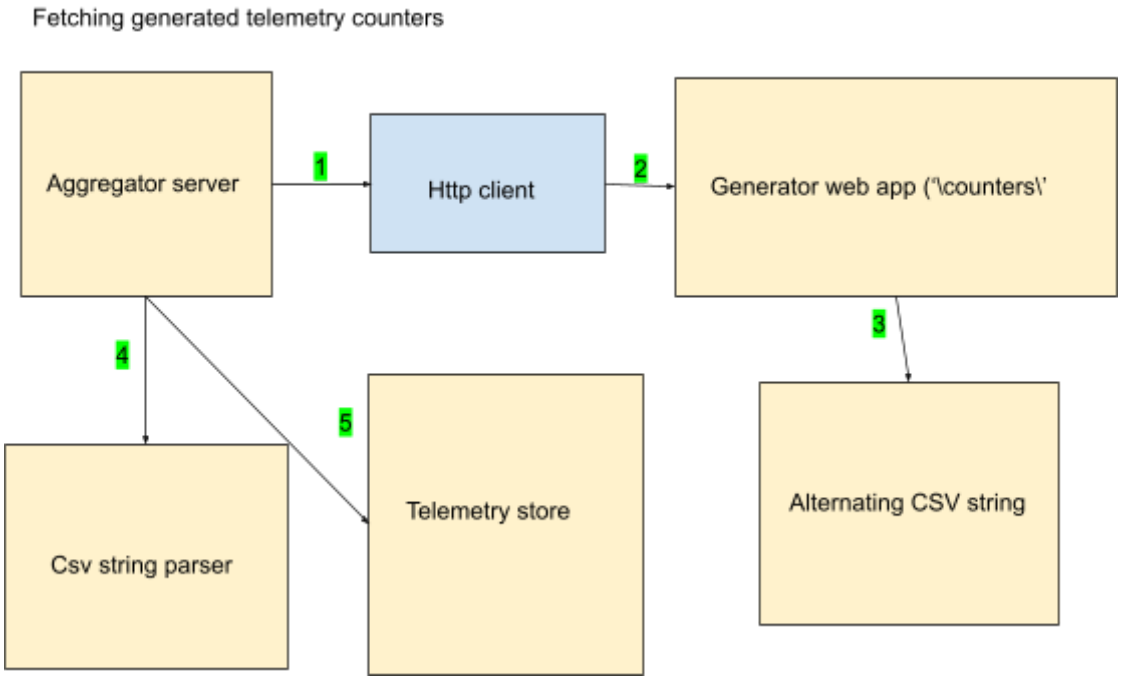
c. Performance:
    i. Receive request on endpoint method
    ii. Access performance store to get all metrics (for all servers) with no locking/blocking issue.
    iii. Reply as json

4. Performance update:
    a. Ingest:
        i. On start of each ingest take current time
        ii. On end of each ingest take current time
        iii. Update performance store with time delta
        iv. Update performance store ingest counter success/failure

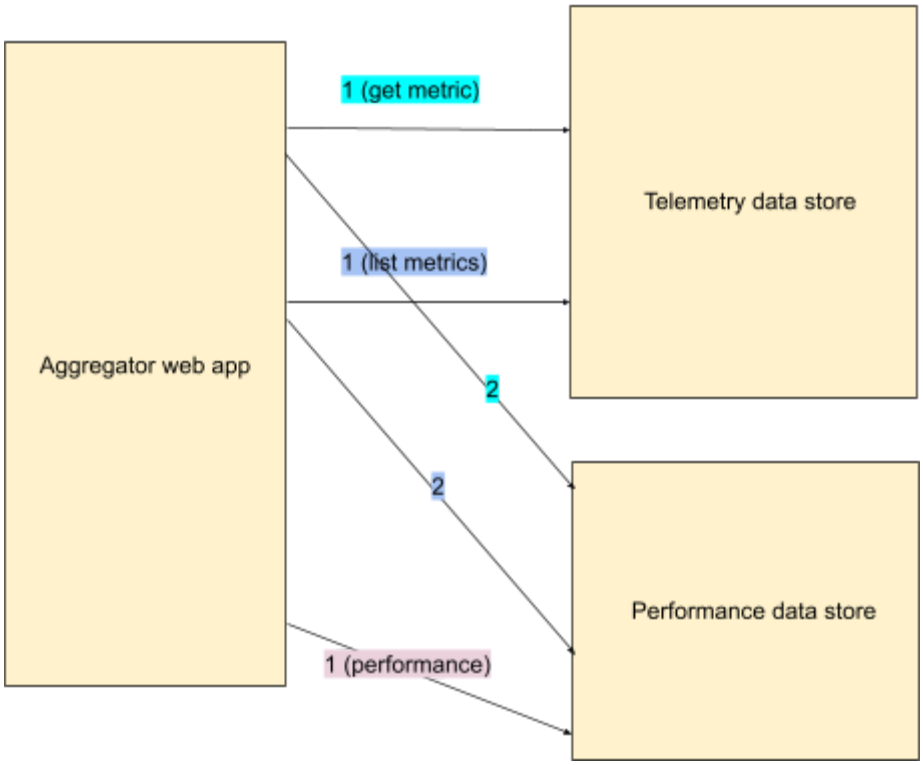
    b. api:
        i. On start of each api (list_metrics, get_metric) take current time
        ii. On end of each api (list_metrics, get_metric) take current time
        iii. Update performance store with time delta for api
        iv. Update performance store api counter success/failure

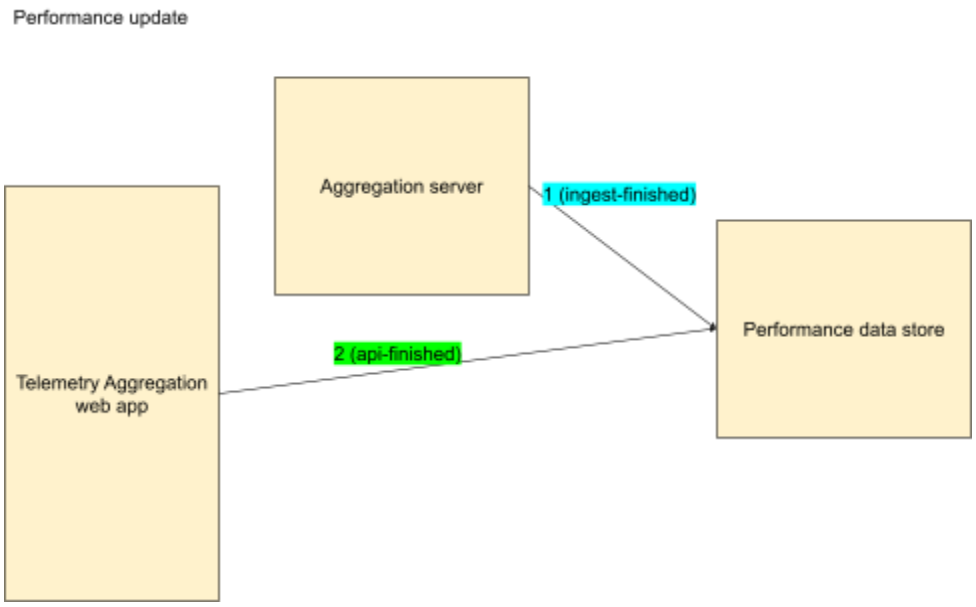
HLD


Generate telemetry counters
Telemetry Generator
Metric Generator
Alternating CSV build/save


Fetching generated telemetry counters
Aggregator server
1
Http client
2
Generator web app (‘\counters\’
3
4
5
Csv string parser
Telemetry store
Alternating CSV string

Serving Telemetry data





Error Handling

- Ensuring improved error handling by answering any possible concrete exception - with minimum of log report (and performance counters update wherever applicable) preferably with concrete handlers (alternatives).
- Minimize general exception handling
- Separate data from behaviour allowing objects resurrection.

Opens (suggestions - abstract)

Scalability

- Enable scaling writes vs scaling reads separation consider CQRS pattern

- Sharding data based on switches/server names for scaling writes (will need a shared memory system for reads) may enable scaling writes with hash based scaling (e.g. consistent hashing).
- Horizontal scaling utilizing containerization triggering scale based on external resources (CPU/Memory/etc.)

- API gateway: Applying different replicas to server different APIs based on usage statistics.

- Load balancing (layer 7): allowing scaling reads via hash based scaling (e.g. consistent hashing) - multiple replicas each serving other users based on client

details (regional).

- Load balancing (layer 7): allowing scaling writes via hash based scaling on the counters endpoint (e.g. consistent hashing) - multiple replicas each serving different groups of switches real/generated telemetry counters data.

Time/Space Efficiency

- Consider minimum object run time re construction and utilize existing
- Design objects with memory aligned to PyMalloc allocations - avoiding system memory allocation
- Minimize locks
- Apply pre failures checks

Fault-Tolerance and Resiliency

- Ensure fine tuned timeouts
- Ensure fallbacks (default coherent response)
- Implement retries w/o exponential backoffs
- Implement circuit breaker for graceful service restoration
- Implement rate limiting for avoidance

Higher throughput and Performance

- Concurrency model must consider short messages (e.g. performance/get metric) request utilizing ***asyncio***, and possible threads pools, jobs and pagination for long message (e.g. list metrics)
- Utilize multiprocessing to utilize multiple CPU cores

Persistancy

- No requirements

High Availability

- Utilize redundant replicas