# TS Metrics Design

**Author**: Li Ning
**Date**: 06/13/2022



## Problem Statement

- Metrics diverge

TorchServe has two mechanisms to emit metrics.

1. Emit metrics to logs files in a StatsD like format by default

In this case, all metrics (ie. both frontend and backend metrics) are recorded in log files. Users have to write a regex to parse the log file to build a dashboard.

2. Emit Prometheus formatted metrics

In this case, only 3 metrics from frontend are emitted via metrics endpoint.
- ts_inference_requests_total
- ts_inference_latency_microseconds
- ts_queue_latency_microseconds

- No Metric Type

TorchServe metrics log is not standard StatsD format. There is no metrics type such as counter, gauge or timer. Except the metrics in case 2, most metrics are raw data, not real time series

metric log. Users are not able to easily build a dashboard via third-party software such as splunk, elastic.

- No central metrics definition

There is no central place to collect all metrics definitions. It is difficult not only for a new backend to keep consistent metrics, but also for users to know the available metrics in TorchServe.

- Metrics delay

TorchServe frontend scans, parses backend logs, and then writes model metric logs. By nature, this mechanism can cause metrics logs to be delayed.

## Solution

### Architecture

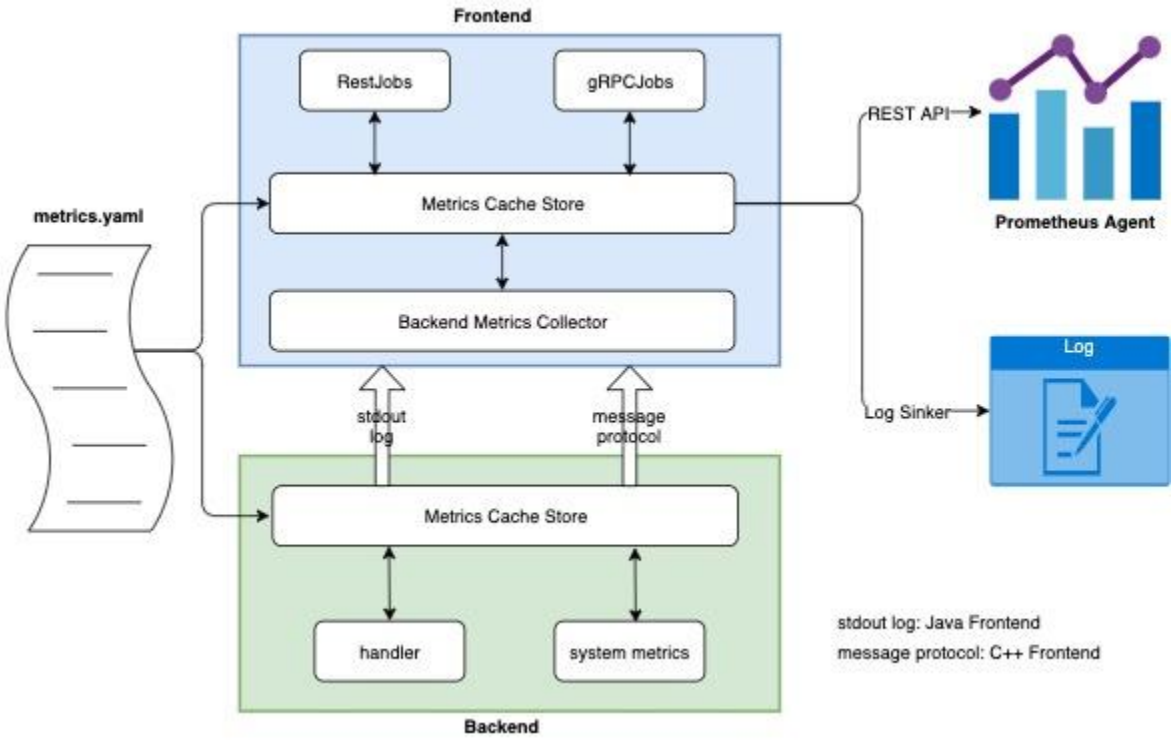


Figure 1: TorchServe Metrics Architecture

### Metrics Configuration

TorchServe defines metrics in a yaml file metrics.yaml. It includes frontend metrics (ie. ts_metrics) and backend metrics (ie. model_metrics). The metrics definition is loaded in both frontend and backend cache separately when TorchServe is started. The format is defined as the following. Here, we use the Prometheus metric type terminology which can match or apply all the other metrics systems.

```yaml
# mode is used by frontend
# mode options: tslog or prometheus
mode: tslog

dimensions:
  model_name: &model_name "model"
  host: &host "host"

ts_metrics:
  counter:
    - name:
      unit: ms
      dimensions: [*model_name, *host]


  gauge:
    - name:
      unit: ms
      dimensions: [*model_name, *host]

  histogram:
    - name:
      unit: ms
      dimensions: [*model_name, *host]

model_metrics:
  counter:
    - name:
      unit: ms
      dimensions: [*model_name, *host]

  gauge:
    - name:
      unit: ms
      dimensions: [*model_name, *host]
  histogram:
    - name:
      unit: ms
      dimensions: [*model_name, *host]
```

### Metrics Interface

- Frontend

TorchServe frontend supports Prometheus client. Frontend automatically caches all the metrics defined in metrics.yaml (ie. including  ts_metrcis and model_metrics) as Prometheus metrics objects. There are three operation APIs.

```
MetricsCaching {
  Counter getCounter(metricName, dimensions);
  Gauge getGauge(metricName, dimensions);
  Histogram getHistogram(metricName, dimensions);
}
```

- Backend

TorchServe backend can either be Python processes or be C++ processes. According to the incoming model registration requests, these processes are dynamically created by the frontend. It is impossible to provide a static config for Prometheus pull mode. As for Prometheus push mode, it is suited for ephemeral and batch jobs, and not supported on AWS cloudwatch.

So TorchServe backend defines its own metrics APIs as the following. Each process caches all of the model metrics defined in metrics.yaml, and emits the metrics from cache to frontend. Frontend will generate Prometheus metrics for backend raw data points. Essentially, frontend is the central place to generate Prometheus metrics.

```
MetricsCaching {
  // create a new Metric and add into cache
  Metric addMetric(metricName, unit, dimensions, metricType);

  // get a Metric from cache
  Metric getMetric(metricName, dimensions);

 // emit all metrics to frontend and reset all metrics value to 0
 void flush();
}

Class Metric {
  Metric(metricName, unit, dimensions, metricType);

  // update the metric value
  void addOrUpdate(value);
  void addOrUpdate(value, requestId);

  // reset value to 0
  void reset();
```

```
}

Interface MetricsEmitter {
  // emit the metric to frontend channel
  // the channel can be logger, SHM, queue etc.
  emit(Metric);
}
```

### Customer Metrics Support

Customers are able to add customized metrics in metrics.yaml. Existing customer metrics API will be a legacy API definition which still needs to be supported in the existing Python backend.

New customers can use the new metrics interface to create new metrics.

- Frontend
  step1: Add metric definition in metrics.yaml
  step2: Get a metric via MetricsCaching.getXXXX()
  step3: Prometheus metric operation

- Backend handler
  step1: Add metric definition in metrics.yaml
  step2: Create a metric via MetricsCaching.addMetric
  step3: Get a metric via MetricsCaching.getMetric
  step4: metric operation addOrUpdate

### Metrics Caching

TorchServe caches metrics in the frontend and backend. Backend flushes metrics cache once a load model or inference request is done. Frontend aggregates or updates backend metrics via Prometheus client API.

### Frontend Metrics Collector

Frontend needs to collect backend and system  metrics. Current TorchServe frontend extracts these  metrics by scanning backend stdout log in the following two scenarios:

1. System metrics: Frontend MetricsCollector periodically starts the Python process and scans its stdout.

2. Backend metrics: Frontend each worker extracts metrics by scanning backend process stdout log.

The existing solution is simple, but expensive. This can be a solution for Java frontend + C++ backend too.

In the long term, a message protocol b/w frontend and backend needs to be redefined in C++ frontend and backend.

### Metrics Permanent Collector

TorchServe supports two ways to store metrics permanently.

- Metrics Log

By default, TorchServe stores all metrics raw data in StatsD like format. That is,

2022-03-30T10:58:39,358 - QueueTime.ms:1|#Level:Host|#hostname:147dda19895c.ant.amazon.com,timestamp:1648663119

2022-03-30T10:58:35,645 - MemoryUsed.Megabytes:7832.8125|#Level:Host|#hostname:147dda19895c.ant.amazon.com,timestamp:1648663115

2022-03-30T10:58:39,357 - HandlerTime.Milliseconds:581.7|#ModelName:benchmark,Level:Model|#hostname:147dda19895c.ant.amazon.com,requestID:75993b11-281a-4550-88b6-aa1a5cba0d8f,timestamp:1648663119

2022-03-30T10:58:39,357 - PredictionTime.Milliseconds:581.95|#ModelName:benchmark,Level:Model|#hostname:147dda19895c.ant.amazon.com,requestID:75993b11-281a-4550-88b6-aa1a5cba0d8f,timestamp:1648663119

- Prometheus

TorchServe metrics can be pulled into Prometheus via HTTP collector defined in config file prometheus.yaml. For example:

```
global:
  scrape_interval:     15s
  evaluation_interval: 15s

scrape_configs:
  - job_name: 'prometheus'
```

```
    static_configs:
    - targets: ['localhost:9090']
  - job_name: 'torchserve'
    static_configs:
    - targets: ['localhost:8082'] #TorchServe metrics endpoint
```

## Existing Metrics

By default, existing TorchServe metrics includes two parts:

- ts_metrics

| MetricName | Unit | Dimensions | Comments |
|---|---|---|---|
| Requests2XX | Count | Level:host | value: "host" |
| Requests4XX | Count | "Level" | value: "host" |
| Requests5XX | Count | "Level" | value: "host" |
| QueueTime | ms | "Level" | value: "host" |
| worker_name | ms | "Level" | create worker thread timestamp; invalid metric definition |
| WorkerThreadTime | ms | "Level" | cmd response latency b/w worker thread and backend |
| ts_inference_requests_total | | "uuid", "model_name", "model_version" | not available in ts_metrics.log; "uuid" is invalid |
| ts_inference_latency_microseconds | | "uuid", "model_name", "model_version" | - not available in ts_metrics.log;<br>- only available in REST API |
| ts_queue_latency_microseconds | | "uuid", "model_name", "model_version" | - represent as "QueueTime" in ts_metrics.log;<br>- only available in REST API |
| CPUUtilization | Percent | "Level" | value: "host" |
| MemoryUsed | Megabytes | "Level" | value: "host" |
| MemoryAvailable | Megabytes | "Level" | value: "host" |
| MemoryUtilization | Percent | "Level" | value: "host" |
| DiskUsage | Gigabytes | "Level" | value: "host" |

| | | | |
|---|---|---|---|
| DiskUtilization | Percent | "Level" | value: "host" |
| DiskAvailable | Gigabytes | "Level" | value: "host" |
| GPUMemoryUtilization | Percent | "Level", "device_id" | |
| GPUMemoryUsed | Megabytes | "Level", "device_id" | |
| GPUUtilization | Percent | "Level", "device_id" | |

- model_metrics

| **MetricName** | **Unit** | **Dimensions** | **Comments** |
|---|---|---|---|
| PredictionTime | Milliseconds | "ModelName", "Level" | |

## Implementation Key Points and Pseudo Code

This section is to summarize the key points in implementation and give pseudo code.

- ### Metrics Configuration
  1. In the metics configuration file,the "dimensions" node is to provide a list of the **dimension name aliases** by using yaml alias feature, **not** a list of **dimension values.**
  2. Default metrics configuration file path should be the same as default config.properties file path. In other words, it should be packed via Java frontend gradle.
  3. Frontend passes metrics configuration file path to backend via command line.

- ### Metrics API

  A metrics object is created based on the following parameters:
  1. metric type
  2. metric name
  3. dimension **names**, **not** dimension **values**
  4. unit

```
Interface IMetric {
  IMetric(type, name, unit, dimension_names);
```

```
  // update the metric value
  virtual void addOrUpdate(dimension_values, value);
}

// mode: tslog
class TSLogMetric implements IMetric { // Used by frontend or backend

  void addOrUpdate(dimension_values, value) {
    // update values
    // write log if it is needed
    // reset value if it is needed
  }

  void addOrUpdate(dimension_values, requestId, value) {
    // update values
    // write log if it is needed
    // reset value if it is needed
  }

  // for python backend backward compatible
  void addOrUpdate(value) {
    // update values
    // write log if it is needed
    // reset value if it is needed
  }

  // for python backend backward compatible
  void addOrUpdate(requestId, value) {
    // update values
    // write log if it is needed
    // reset value if it is needed
  }

  Map<dimension_values, MValue> values;

  class MValue {
    requestId;
    value
  }
}

// mode: prometheus
class PrometheusCounter implements IMetric { // Used by frontend
```

```
  void addOrUpdate(dimension_values, value);
}
class PrometheusGauge implements IMetric { // Used by frontend
  void addOrUpdate(dimension_values, value);
}
class PrometheusHistogram implements IMetric { // Used by frontend
  void addOrUpdate(dimension_values, value);
}
class PrometheusSummary implements IMetric { // Used by frontend
  void addOrUpdate(dimension_values, value);
}
```

- ### Caching

```
MetricsCaching {
  // create a new Metric and add into cache
  IMetric addMetric(type, name, unit, dimension_names);

  // get a Metric from cache
  IMetric getMetric(type, metricName);

  Map<MetricType, Map<MetricName, IMetric>> cache;
}
```

- ### Python Backend Backward Compatible

To support existing python backend metrics API, metrics store needs to be migrated to the caching solution.

- ### Workflow

1. Frontend or Backend Initialization phase
    a. parse metrics configuration file
    b. add metric objects into MetricsCaching

2. Frontend or Backend Runtime phase
    a. Get a metric from metrics cache
    b. Update the metric