PROMETHEUS REMOTE-WRITE SPECIFICATION

- Version: 1.0
- Status: Published
- Date: April 2023

This document is intended to define and standardise the API, wire format, protocol and semantics of the existing, widely and organically adopted protocol, and not to propose anything new.

The remote write specification is intended to document the standard for how Prometheus and Prometheus remote-write-compatible agents send data to a Prometheus or Prometheus remote-write compatible receiver.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 (https://datatracker.ietf.org/doc/html/rfc2119).

Introduction

Background

The remote write protocol is designed to make it possible to reliably propagate samples in real-time from a sender to a receiver, without loss.

The remote write protocol is designed to make stateless implementations of the server possible; as such there are little-to-no inter-message references. As such the protocol is not considered "streaming." To achieve a streaming effect multiple messages should be sent over the same connection using e.g. HTTP/1.1 or HTTP/2. "Fancy" technologies such as gRPC were considered, but at the time were not widely adopted, and it was challenging to expose gRPC services to the internet behind load balancers such as an AWS EC2 ELB.

The remote write protocol contains opportunities for batching, e.g. sending multiple samples for different series in a single request. It is not expected that multiple samples for the same series will be commonly sent in the same request, although there is support for this in the protocol.

The remote write protocol is not intended for use by applications to push metrics to Prometheus remote-write-compatible receivers. It is intended that a Prometheus remote-write-compatible sender scrapes instrumented applications or exporters and sends remote write messages to a server.

A test suite can be found at https://github.com/prometheus/compliance/tree/main/remote_write_sender (https://github.com/prometheus/compliance/tree/main/remote_write_sender).

Glossary

For the purposes of this document the following definitions MUST be followed:

- a "Sender" is something that sends Prometheus Remote Write data.
- a "Receiver" is something that receives Prometheus Remote Write data.
- a "Sample" is a pair of (timestamp, value).
- a "Label" is a pair of (key, value).
- a "Series" is a list of samples, identified by a unique set of labels.

Definitions

Protocol

The Remote Write Protocol MUST consist of RPCs with the following signature:

```
func Send(WriteRequest)

message WriteRequest {
  repeated TimeSeries timeseries = 1;
  // Cortex uses this field to determine the source of the write request.
  // We reserve it to avoid any compatibility issues.
  reserved  2;

  // Prometheus uses this field to send metadata, but this is
  // omitted from v1 of the spec as it is experimental.
  reserved  3;
}

message TimeSeries {
  repeated Label labels   = 1;
  repeated Sample samples = 2;
}

message Label {
  string name  = 1;
  string value = 2;
}

message Sample {
  double value    = 1;
  int64 timestamp = 2;
}
```

Remote write Senders MUST encode the Write Request in the body of a HTTP POST request and send it to the Receivers via HTTP at a provided URL path. The Receiver MAY specify any HTTP URL path to receive metrics.

Timestamps MUST be int64 counted as milliseconds since the Unix epoch. Values MUST be float64.

The following headers MUST be sent with the HTTP request:

- `Content-Encoding: snappy`
- `Content-Type: application/x-protobuf`
- `User-Agent: <name & version of the sender>`
- `X-Prometheus-Remote-Write-Version: 0.1.0`

Clients MAY allow users to send custom HTTP headers; they MUST NOT allow users to configure them in such a way as to send reserved headers. For more info see https://github.com/prometheus/prometheus/pull/8416 (https://github.com/prometheus/prometheus/pull/8416).

The remote write request in the body of the HTTP POST MUST be compressed with Google's Snappy (https://github.com/google/snappy). The block format MUST be used - the framed format MUST NOT be used.

The remote write request MUST be encoded using Google Protobuf 3, and MUST use the schema defined above. Note the Prometheus implementation (https://github.com/prometheus/prometheus/blob/v2.24.0/prompb/remote.proto) uses gogoproto optimisations (https://github.com/gogo/protobuf) - for receivers written in languages other than Golang the gogoproto types MAY be substituted for line-level equivalents.

The response body from the remote write receiver SHOULD be empty; clients MUST ignore the response body. The response body is RESERVED for future use.

Backward and forward compatibility

The protocol follows semantic versioning 2.0 (https://semver.org/): any 1.x compatible receivers MUST be able to read any 1.x compatible sender and so on. Breaking/backwards incompatible changes will result in a 2.x version of the spec.

The proto format itself is forward / backward compatible, in some respects:

- Removing fields from the proto will mean a major version bump.
- Adding (optional) fields will be a minor version bump.

Negotiation:

- Senders MUST send the version number in a headers.
- Receivers MAY return the highest version number they support in a response header ("X-Prometheus-Remote-Write-Version").
- Senders who wish to send in a format >1.x MUST start by sending an empty 1.x, and see if the response says the receiver supports something else. The Sender MAY use any supported version . If there is no version header in the response, senders MUST assume 1.x compatibility only.

Labels

The complete set of labels MUST be sent with each sample. Whatsmore, the label set associated with samples:

- SHOULD contain a `__name__` label.
- MUST NOT contain repeated label names.
- MUST have label names sorted in lexicographical order.
- MUST NOT contain any empty label names or values.

Senders MUST only send valid metric names, label names, and label values:

- Metric names MUST adhere to the regex `[a-zA-Z_:]([a-zA-Z0-9_:])*`.
- Label names MUST adhere to the regex `[a-zA-Z_]([a-zA-Z0-9_])*`.
- Label values MAY be any sequence of UTF-8 characters .

Receivers MAY impose limits on the number and length of labels, but this will be receiver-specific and is out of scope for this document.

Label names beginning with "__" are RESERVED for system usage and SHOULD NOT be used, see Prometheus Data Model (https://prometheus.io/docs/concepts/data_model/).

Remote write Receivers MAY ingest valid samples within a write request that otherwise contains invalid samples. Receivers MUST return a HTTP 400 status code ("Bad Request") for write requests that contain any invalid samples. Receivers SHOULD provide a human readable error message in the response body. Senders MUST NOT try and interpret the error message, and SHOULD log it as is.

Ordering

Prometheus Remote Write compatible senders MUST send samples for any given series in timestamp order. Prometheus Remote Write compatible Senders MAY send multiple requests for different series in parallel.

Retries & Backoff

Prometheus Remote Write compatible senders MUST retry write requests on HTTP 5xx responses and MUST use a backoff algorithm to prevent overwhelming the server. They MUST NOT retry write requests on HTTP 2xx and 4xx responses other than 429. They MAY retry on HTTP 429 responses, which could result in senders "falling behind" if the server cannot keep up. This is done to ensure data is not lost when there are server side errors, and progress is made when there are client side errors.

Prometheus remote Write compatible receivers MUST respond with a HTTP 2xx status code when the write is successful. They MUST respond with HTTP status code 5xx when the write fails and SHOULD be retried. They MUST respond with HTTP status code 4xx when the request is invalid, will never be able to succeed and should not be retried.

Stale Markers

Prometheus remote write compatible senders MUST send stale markers when a time series will no longer be appended to.

Stale markers MUST be signalled by the special NaN value 0x7ff0000000000002. This value MUST NOT be used otherwise.

Typically the sender can detect when a time series will no longer be appended to using the following techniques:

1. Detecting, using service discovery, that the target exposing the series has gone away
2. Noticing the target is no longer exposing the time series between successive scrapes
3. Failing to scrape the target that originally exposed a time series
4. Tracking configuration and evaluation for recording and alerting rules

Out of Scope

This document does not intend to explain all the features required for a fully Prometheus-compatible monitoring system. In particular, the following areas are out of scope for the first version of the spec:

The "up" metric The definition and semantics of the "up" metric are beyond the scope of the remote write protocol and should be documented separately.

HTTP Path The path for HTTP handler can be anything - and MUST be provided by the sender. Generally we expect the whole URL to be specified in config.

Persistence It is recommended that Prometheus Remote Write compatible senders should persistently buffer sample data in the event of outages in the receiver.

Authentication & Encryption as remote write uses HTTP, we consider authentication & encryption to be a transport-layer problem. Senders and receivers should support all the usual suspects (Basic auth, TLS etc) and are free to add potentially custom authentication options. Support for custom authentication in the Prometheus remote write sender and eventual agent should not be assumed, but we will endeavour to support common and widely used auth protocols, where feasible.

Remote Read this is a separate interface that has already seen some iteration, and is less widely used.

Sharding the current sharding scheme in Prometheus for remote write parallelisation is very much an implementation detail, and isn't part of the spec. When senders do implement parallelisation they MUST preserve per-series sample ordering.

Backfill The specification does not place a limit on how old series can be pushed, however server/implementation specific constraints may exist.

Limits Limits on the number and length of labels, batch sizes etc are beyond the scope of this document, however it is expected that implementation will impose reasonable limits.

Push-based Prometheus Applications pushing metrics to Prometheus Remote Write compatible receivers was not a design goal of this system, and should be explored in a separate doc.

Labels Every series MAY include a "job" and/or "instance" label, as these are typically added by service discovery in the Sender. These are not mandatory.

Future Plans

This section contains speculative plans that are not considered part of protocol specification, but are mentioned here for completeness.

Transactionality Prometheus aims at being "transactional" - i.e. to never expose a partially scraped target to a query. We intend to do the same with remote write - for instance, in the future we would like to "align" remote write with scrapes, perhaps such that all the samples, metadata and exemplars for a single scrape are sent in a single remote write request. This is yet to be designed.

Metadata and Exemplars In line with above, we also send metadata (type information, help text) and exemplars along with the scraped samples. We plan to package this up in a single remote write request - future versions of the spec may insist on this. Prometheus currently has experimental support for sending metadata and exemplars.

Optimizations We would like to investigate various optimizations to reduce message size by eliminating repetition of label names and values.

Related

Compatible Senders and Receivers

The spec is intended to describe how the following components interact:

- Prometheus (https://github.com/prometheus/prometheus/tree/master/storage/remote) (as both a "sender" and a "receiver")
- Avalanche (https://github.com/prometheus-community/avalanche) (as a "sender") - A Load Testing Tool Prometheus Metrics.
- Cortex (https://github.com/cortexproject/cortex/blob/master/pkg/util/push/push.go#L20) (as a "receiver")
- Elastic Agent (https://docs.elastic.co/integrations/prometheus#prometheus-server-remote-write) (as a "receiver")
- Grafana Agent (https://github.com/grafana/agent) (as both a "sender" and a "receiver")
- GreptimeDB (https://github.com/greptimeTeam/greptimedb) (as a "receiver" (https://docs.greptime.com/user-guide/write-data/prometheus#prometheus))
- InfluxData's Telegraf agent. (as a sender (https://github.com/influxdata/telegraf/tree/master/plugins/serializers/prometheusremotewrite), and as a receiver (https://github.com/influxdata/telegraf/pull/8967))
- M3 (https://m3db.io/docs/integrations/prometheus/#prometheus-configuration) (as a "receiver")
- Mimir (https://github.com/grafana/mimir) (as a "receiver")
- OpenTelemetry Collector (https://github.com/open-telemetry/opentelemetry-collector-releases/) (as a "sender" (https://github.com/open-telemetry/opentelemetry-collector-contrib/tree/main/exporter/prometheusremotewriteexporter#readme) and eventually as a "receiver")
- Thanos (https://thanos.io/tip/components/receive.md/) (as a "receiver")
- Vector (as a "sender" (https://vector.dev/docs/reference/configuration/sinks/prometheus_remote_write/) and a "receiver" (https://vector.dev/docs/reference/configuration/sources/prometheus_remote_write/))
- VictoriaMetrics (https://github.com/VictoriaMetrics/VictoriaMetrics) (as a "receiver" (https://docs.victoriametrics.com/#prometheus-setup))

FAQ

Why did you not use gRPC? Funnily enough we initially used gRPC, but switched to Protos atop HTTP as in 2016 it was hard to get them past ELBs: https://github.com/prometheus/prometheus/issues/1982 (https://github.com/prometheus/prometheus/issues/1982)

Why not streaming protobuf messages? If you use persistent HTTP/1.1 connections, they are pretty close to streaming… Of course headers have to be re-sent, but yes that is less expensive than a new TCP set up.

Why do we send samples in order? The in-order constraint comes from the encoding we use for time series data in Prometheus, the implementation of which is append only. It is possible to remove this constraint, for instance by buffering samples and reordering them before encoding. We can investigate this in future versions of the protocol.

How can we parallelise requests with the in-order constraint? Samples must be in-order *for a given series*. Remote write requests can be sent in parallel as long as they are for different series. In Prometheus, we shard the samples by their labels into separate queues, and then writes happen sequentially in each queue. This guarantees samples for the same series are delivered in order, but samples for different series are sent in parallel - and potentially "out of order" between different series.

We believe this is necessary as, even if the receiver could support out-of-order samples, we can't have agents sending out of order as they would never be able to send to Prometheus, Cortex and Thanos. We’re doing this to ensure the integrity of the ecosystem and to prevent confusing/forking the community into "prometheus-agents-that-can-write-to-prometheus" and those that can’t.

This documentation is open-source (https://github.com/prometheus/docs#contributing-changes). Please help improve it by filing issues or pull requests.

© Prometheus Authors 2014-2024 | Documentation Distributed under CC-BY-4.0

© 2024 The Linux Foundation. All rights reserved. The Linux Foundation has registered trademarks and uses trademarks. For a list of trademarks of The Linux Foundation, please see our Trademark Usage (https://www.linuxfoundation.org/trademark-usage) page.