Prometheus Remote-Write 2.0 specification [EXPERIMENTAL]

- Introduction
  - Background
  - Glossary
- Definitions
  - Protocol
    - Content-Encoding
    - Content-Type
    - X-Prometheus-Remote-Write-Version
    - User-Agent
  - Response
    - Required Written Response Headers
    - Partial Write
    - Unsupported Request Content
    - Invalid Samples
  - Retries & Backoff
    - Retries on Partial Writes
  - Backward and Forward Compatibility
    - 2.x vs 1.x Compatibility
- Protobuf Message
  - io.prometheus.write.v2.Request
    - Symbols
    - Series Labels
    - Samples and Histogram Samples

    - Metadata
    - Exemplars
    - Modes
        - stateless Mode
        - stateless-st Mode
- Out of Scope
- Future Plans
- Related
    - FAQ

- Version: 2.0-rc.6
- Status: **Experimental**
- Date: October 2026

The Remote-Write specification, in general, is intended to document the standard for how Prometheus and Prometheus Remote-Write compatible senders send data to Prometheus or Prometheus Remote-Write compatible receivers.

This document is intended to define a second version of the Prometheus Remote-Write API with minor changes to protocol and semantics. This second version adds a new Protobuf Message with new features enabling more use cases and wider adoption on top of performance and cost savings. The second version also deprecates the previous Protobuf Message from a 1.0 Remote-Write specification and adds mandatory `X-Prometheus-Remote-Write-*-Written` HTTP response headers for reliability purposes. Finally, this spec outlines how to implement backwards-compatible senders and receivers (even under a single endpoint) using existing basic content negotiation request headers. More advanced, automatic content negotiation mechanisms might come in a future minor version if needed. For the rationales behind the 2.0 specification, see the formal proposal.

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.

> NOTE: This is a release candidate for Remote-Write 2.0 specification. This means that this specification is currently in an experimental state--no major changes are expected, but we reserve the right to break the compatibility if it's necessary, based on the early adopters' feedback. The potential feedback, questions and suggestions should be added as comments to the PR with the

> open proposal.

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 be stateless; there is strictly no inter-message communication. 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 Protobuf Message.

Compliance tests can be found at:

- sender: https://github.com/prometheus/compliance/tree/main/remotewrite/sender
- receiver: https://github.com/prometheus/compliance/tree/main/remotewrite/receiver

Glossary

In this document, the following definitions are followed:

- `Remote-Write` is the name of this Prometheus protocol.
- a `Protocol` is a communication specification that enables the client and server to transfer metrics.
- a `Protobuf Message` (or Proto Message) refers to the content type definition of the data structure for this Protocol. Since the specification uses Google Protocol Buffers ("protobuf") exclusively, the schema is defined in a "proto" file and represented by a single Protobuf "message".
- a `Wire Format` is the format of the data as it travels on the wire (i.e. in a network). In the case of Remote-Write, this is always the compressed binary protobuf format.

- a `Sender` is something that sends Remote-Write data.
- a `Receiver` is something that receives (writes) Remote-Write data. The meaning of `Written` is up to the Receiver e.g. usually it means storing received data in a database, but also just validating, splitting or enhancing it.
- a `Mode` is a set of additional rules on top of `Protocol` that tightens optional sender and receiver semantics into a deterministic contract.
- `Written` refers to data the `Receiver` has received and is accepting. Whether or not it has ingested this data to persistent storage, written it to a WAL, etc. is up to the `Receiver`. The only distinction is that the `Receiver` has accepted this data rather than explicitly rejecting it with an error response.
- a `Sample` is a triplet of (start timestamp, timestamp, value).
- a `Histogram` is a triplet of (start timestamp, timestamp, histogram value).
- a `Label` is a pair of (key, value).
- a `Series` is a list of samples (or histograms), identified by a unique set of labels.

Definitions

Protocol

The Remote-Write Protocol MUST consist of RPCs with the request body serialized using a Google Protocol Buffers and then compressed.

The protobuf serialization MUST use either of the following Protobuf Messages:

- The `prometheus.WriteRequest` introduced in the Remote-Write 1.0 specification. As of 2.0, this message is deprecated. It SHOULD be used only for compatibility reasons. Senders and Receivers MAY NOT support the `prometheus.WriteRequest`.
- The `io.prometheus.write.v2.Request` introduced in this specification and defined below. Senders and Receivers SHOULD use this message when possible. Senders and Receivers MUST support the `io.prometheus.write.v2.Request`.

Protobuf Message MUST use binary Wire Format. Then, MUST be compressed with Google's Snappy. Snappy's block format MUST be used -- the framed format MUST NOT be used.

Senders MUST send a serialized and compressed Protobuf Message in the body of an HTTP POST request and send it to the Receiver via HTTP at the provided URL path. Receivers MAY specify any HTTP URL path to receive metrics.

Senders MUST send the following reserved headers with the HTTP request:

- `Content-Encoding`

- `Content-Type`
- `X-Prometheus-Remote-Write-Version`
- `User-Agent`

Senders MAY allow users to add custom HTTP headers; they MUST NOT allow users to configure them in such a way as to send reserved headers.

Content-Encoding

```
Content-Encoding: <compression>
```

Content encoding request header MUST follow the RFC 9110. Senders MUST use the `snappy` value. Receivers MUST support `snappy` compression. New, optional compression algorithms might come in 2.x or beyond.

Content-Type

```
Content-Type: application/x-protobuf
Content-Type: application/x-protobuf;proto=<fully qualified name>
Content-Type: application/x-protobuf;proto=<fully qualified name>;mode=<mode>
```

Content type request header MUST follow the RFC 9110. Senders MUST use `application/x-protobuf` as the only media type. Senders MAY add `;proto=` parameter to the header's value to indicate the fully qualified name of the Protobuf Message that was used, from the two mentioned above.

Senders MAY also add `;mode=` parameter to indicate the mode (additional semantic rules) for the Protobuf Message, if any is defined for it. As a result, Senders MUST send one of the following supported header values:

For the deprecated message introduced in PRW 1.0, identified by `prometheus.WriteRequest`:

- `Content-Type: application/x-protobuf`
- `Content-Type: application/x-protobuf;proto=prometheus.WriteRequest`

For the message introduced in PRW 2.0, identified by `io.prometheus.write.v2.Request`:

- `Content-Type: application/x-protobuf;proto=io.prometheus.write.v2.Request`
- `Content-Type: application/x-protobuf;proto=io.prometheus.write.v2.Request;mode=none` (equivalent to omitting `;mode=`; see Modes)
- `Content-Type: application/x-protobuf;proto=io.prometheus.write.v2.Request;mode=stateless` (see Modes)
- `Content-Type: application/x-protobuf;proto=io.prometheus.write.v2.Request;mode=stateless-st` (see Modes)

When talking to 1.x Receivers, Senders SHOULD use `Content-Type: application/x-protobuf` for backward compatibility. Otherwise, Senders SHOULD use `Content-Type: application/x-protobuf;proto=io.prometheus.write.v2.Request` (optionally with `;mode=none`, `;mode=stateless`, or `;mode=stateless-st`). More Protobuf Messages or modes might come in 2.x or beyond.

Receivers MUST use the content type header to identify the Protobuf Message schema to use, and MAY use it to identify the mode. Accidental wrong schema choices may result in non-deterministic behaviour (e.g. corruptions).

> NOTE: Thanks to reserved fields in `io.prometheus.write.v2.Request`, Receiver accidental use of wrong schema with `prometheus.WriteRequest` will result in empty message. This is generally for convenience to avoid surprising errors, but don't rely on it -- future Protobuf Messages might not have this feature.

X-Prometheus-Remote-Write-Version

```
X-Prometheus-Remote-Write-Version: <Remote-Write spec major and minor version>
```

When talking to 1.x Receivers, Senders MUST use `X-Prometheus-Remote-Write-Version: 0.1.0` for backward compatibility. Otherwise, Senders SHOULD use the newest Remote-Write version it is compatible with e.g. `X-Prometheus-Remote-Write-Version: 2.0.0`.

User-Agent

```
User-Agent: <name & version of the Sender>
```

Senders MUST include a user agent header that SHOULD follow the RFC 9110 User-Agent header format.

Response

Receivers that written all data successfully MUST return a success 2xx HTTP status code. In such a successful case, the response body from the Receiver SHOULD be empty and the status code SHOULD be 204 HTTP No Content; Senders MUST ignore the response body. The response body is RESERVED for future use.

Receivers that operate asynchronously (e.g. ingest payloads into a message queue or pipeline for deferred processing without synchronous parsing) MAY return 202 HTTP Accepted status code. This indicates that the request has been accepted for processing, but processing has not been completed.

Receivers MUST NOT return a 2xx HTTP status code, if any of the pieces of sent data known to the Receiver (e.g. Samples, Histograms, Exemplars) were NOT synchronously written successfully (both partial write or full write rejection) or wasn't sent for asynchronous processing (queue full for example). In such a case, the Receiver MUST provide a human-readable error message in the response body. The Receiver's error SHOULD contain information about the amount of the samples being rejected and for what reasons. Senders MUST NOT try and interpret the error message and SHOULD log it as is.

The following subsections specify Sender and Receiver semantics around headers and different write error cases.

Required `Written` Response Headers

Upon a successful content negotiation, Receivers process (write) the received batch of data. Once completed (with success or failure) for each important piece of data (currently Samples, Histograms and Exemplars) Receivers MUST send a dedicated HTTP `X-Prometheus-Remote-Write-*-Written` response header with the precise number of successfully written elements.

Asynchronous Receivers returning `202 Accepted` MAY omit these headers if counts cannot be calculated synchronously (validation is deferred). Asynchronous Receivers SHOULD propagate errors to clients via different channels.

Each header value MUST be a single 64-bit integer. The header names MUST be as follows:

```
X-Prometheus-Remote-Write-Samples-Written <count of all successfully written Samples>
X-Prometheus-Remote-Write-Histograms-Written <count of all successfully written Histogram samples>
X-Prometheus-Remote-Write-Exemplars-Written <count of all successfully written Exemplars>
```

Upon receiving a 2xx status code or upon receiving a 4xx status code, Senders CAN assume that any missing `X-Prometheus-Remote-Write-*-Written` response header means no element from this category (e.g. Sample) was written by the Receiver (count of `0`).

Senders MUST NOT assume the same when:

- Receiving `202 Accepted` status code as asynchronous Receivers may defer validation.
- When using the deprecated `prometheus.WriteRequest` Protobuf Message due to the risk of hitting 1.0 Receiver without this feature.

Senders MAY use those headers to confirm which parts of data were successfully written by the Receiver. Common use cases:

- Better handling of the Partial Write failure situations: Senders MAY use those headers for more accurate client instrumentation and error handling.

- Detecting broken 1.0 Receiver implementations: Senders SHOULD assume 415 HTTP Unsupported Media Type status code when sending the data using `io.prometheus.write.v2.Request` request and receiving a 2xx HTTP status code other than `202 Accepted`, but none of the `X-Prometheus-Remote-Write-*-Written` response headers from the Receiver. This is a common issue for the 1.0 Receivers that do not check the `Content-Type` request header; accidental decoding of the `io.prometheus.write.v2.Request` payload with `prometheus.WriteRequest` schema results in empty result and no decoding errors.
- Detecting other broken implementations or issues: Senders MAY use those headers to detect broken Sender and Receiver implementations or other problems.

Senders MUST NOT assume what Remote Write specification version the Receiver implements from the remote write response headers.

More (optional) headers might come in the future, e.g. when more entities or fields are added and worth confirming.

Partial Write

Senders SHOULD use Remote-Write to send samples for multiple series in a single request. As a result, Receivers MAY write valid samples within a write request that also contains some invalid or otherwise unwritten samples, which represents a partial write case. In such a case, the Receiver MUST return non-2xx status code following the Invalid Samples and Retry on Partial Writes sections.

Unsupported Request Content

Receivers MUST return 415 HTTP Unsupported Media Type status code if they don't support a given content type or encoding provided by Senders. When returning `415 HTTP Unsupported Media Type`, Receivers SHOULD include the `Accept` (listing supported `Content-Type` values, including supported `;proto=` and `;mode=` parameters) and `Accept-Encoding` (listing supported `Content-Encoding` values) response headers as defined in RFC 7694 so Senders can determine what the Receiver supports without guessing.

Senders SHOULD expect 400 HTTP Bad Request for the above reasons from 1.x Receivers, for backwards compatibility.

Invalid Samples

Receivers SHOULD reject data (such as samples, histograms, exemplars, or entire time series) that violates this specification or the negotiated mode. Receivers MAY reject data that violates receiver-specific limits and gaps (such as not supporting native histograms, or label count or length limits). Receivers MUST return a 400 HTTP Bad Request status code for write requests that contain any invalid data (whether rejecting the entire request or performing a partial write) unless the partial

retriable write occurs.

Senders MUST NOT retry on a 4xx HTTP status codes (other than 429), which MUST be used by Receivers to indicate that the write operation will never be able to succeed and should not be retried. Senders MAY retry on the 415 HTTP status code with a different content type or encoding (using the `Accept` and `Accept-Encoding` response headers if present; see Unsupported Request Content) to see if the Receiver supports it.

Retries & Backoff

Receivers MAY return a 429 HTTP Too Many Requests status code to indicate the overloaded server situation. Receivers MAY return the Retry-After header to indicate the time for the next write attempt. Receivers MAY return a 5xx HTTP status code to represent internal server errors.

Senders MAY retry on a 429 HTTP status code. Senders MUST retry write requests on 5xx HTTP. Senders MUST use a backoff algorithm to prevent overwhelming the server. Senders MAY handle the Retry-After response header to estimate the next retry time.

The difference between 429 vs 5xx handling is due to the potential situation of a Sender "falling behind" when the Receiver cannot keep up with the request volume, or the Receiver choosing to rate limit the Sender to protect its availability. As a result, Senders has the option to NOT retry on 429, which allows progress to be made when there are Sender side errors (e.g. too much traffic), while the data is not lost when there are Receiver side errors (5xx).

Retries on Partial Writes

Receivers MAY return a 5xx HTTP or 429 HTTP status code on partial write or partial invalid sample cases when it expects Senders to retry the whole request. In that case, the Receiver MUST support idempotency as Senders MAY retry with the same request.

Backward and Forward Compatibility

The protocol follows semantic versioning 2.0: any 2.x compatible Receiver MUST be able to read any 2.x compatible Senders and vice versa. Breaking or backwards incompatible changes will result in a 3.x version of the spec.

The Protobuf Messages (in Wire Format) themselves are forward / backward compatible, in some respects:

- Removing fields from the Protobuf Message requires a major version bump.

- Adding (optional) fields can be done in a minor version bump.

In other words, this means that future minor versions of 2.x MAY add new optional fields to `io.prometheus.write.v2.Request`, new compressions, Protobuf Messages and negotiation mechanisms, as long as they are backwards compatible (e.g. optional to both Receiver and Sender).

2.x vs 1.x Compatibility

The 2.x protocol is breaking compatibility with 1.x by introducing a new, mandatory `io.prometheus.write.v2.Request` Protobuf Message and deprecating the `prometheus.WriteRequest`.

2.x Senders MAY support 1.x Receivers by allowing users to configure what content type Senders should use. 2.x Senders also MAY automatically fall back to different content types, if the Receiver returns 415 HTTP status code.

Protobuf Message

`io.prometheus.write.v2.Request`

The `io.prometheus.write.v2.Request` references the new Protobuf Message that's meant to replace and deprecate the Remote-Write 1.0's `prometheus.WriteRequest` message.

The full schema and source of the truth is in Prometheus repository in `prompb/io/prometheus/write/v2/types.proto`. The `gogo` dependency and options CAN be ignored (will be removed eventually). They are not part of the specification as they don't impact the serialized format.

The simplified version of the new `io.prometheus.write.v2.Request` is presented below.

```
message Request {
  reserved 1 to 3;

  // symbols contains a de-duplicated array of string elements used for various
  // items in a Request message, like labels and metadata items. For the sender's convenience
  // around empty values for optional fields like unit_ref, symbols array MUST start with
  // empty string.
  //
  // To decode each of the symbolized strings, referenced, by "ref(s)" suffix, you
  // need to lookup the actual string by index from symbols array. The order of
```

```
  // strings is up to the sender. The receiver should not assume any particular encoding.
  repeated string symbols = 4;
  // timeseries represents an array of distinct series with 0 or more samples.
  repeated TimeSeries timeseries = 5;
}

// TimeSeries represents a single series.
message TimeSeries {
  reserved 6;

  // labels_refs is a list of label name-value pair references, encoded
  // as indices to the Request.symbols array. This list's length is always
  // a multiple of two, and the underlying labels should be sorted lexicographically.
  //
  // Note that there might be multiple TimeSeries objects in the same
  // Requests with the same labels e.g. for different exemplars, metadata
  // or start timestamp.
  repeated uint32 labels_refs = 1;

  // TimeSeries messages must specify at least one float sample, native histogram sample
  // or exemplar. Samples and histograms must not be specified at the same time.
  // Exemplars may be specified without samples or histograms. For a typical sender
  // (real-time metric streaming), in healthy cases, there will be only one sample or
histogram.
  //
  // Samples and histograms are sorted by timestamp (older first).
  repeated Sample samples = 2;
  repeated Histogram histograms = 3;

  // exemplars represents an optional set of exemplars for this series or series's
samples/histograms if present.
  repeated Exemplar exemplars = 4;

  // metadata represents the metadata associated with the given series' samples.
  Metadata metadata = 5;
}

// Exemplar contains additional information associated with a series or series's
samples/histograms if present.
// It is typically used to attach an example trace or request ID associated with
```

```
// the metric changes.
message Exemplar {
  // labels_refs is an optional list of label name-value pair references, encoded
  // as indices to the Request.symbols array. This list's len is always
  // a multiple of 2, and the underlying labels should be sorted lexicographically.
  // If the exemplar references a trace it should use the `trace_id` label name, as a best
practice.
  repeated uint32 labels_refs = 1;
  // value represents an exact example value. This can be useful when the exemplar
  // is attached to a histogram, which only gives an estimated value through buckets.
  double value = 2;
  // timestamp represents the timestamp of the exemplar in ms.
  //
  // For Go, see github.com/prometheus/prometheus/model/timestamp/timestamp.go
  // for conversion from/to time.Time to Prometheus timestamp.
  int64 timestamp = 3;
}

// Sample represents series sample.
message Sample {
  // value of the sample.
  double value = 1;
  // timestamp represents timestamp of the sample in ms.
  //
  // For Go, see github.com/prometheus/prometheus/model/timestamp/timestamp.go
  // for conversion from/to time.Time to Prometheus timestamp.
  int64 timestamp = 2;
  // start_timestamp represents an optional start timestamp for the sample,
  // in ms format. This information is typically used for counter, histogram (cumulative)
  // or delta type metrics.
  //
  // For cumulative metrics, the start timestamp represents the time when the
  // counter started counting (sometimes referred to as created timestamp), which
  // can increase the accuracy of certain processing and query semantics (e.g. rates).
  //
  // Note:
  // * That some receivers might require start timestamps for certain metric
  // types; rejecting such samples within the Request as a result.
  // * start timestamp is the same as "created timestamp" name Prometheus used in the past.
  //
```

```
  // For Go, see github.com/prometheus/prometheus/model/timestamp/timestamp.go
  // for conversion from/to time.Time to Prometheus timestamp.
  //
  // Note that the "optional" keyword is omitted due to efficiency and consistency.
  // Zero value means value not set. If you need to use exactly zero value for
  // the timestamp, use 1 millisecond before or after.
  int64 start_timestamp = 3;
}

// Metadata represents the metadata associated with the given series' samples.
message Metadata {
  enum MetricType {
    METRIC_TYPE_UNSPECIFIED    = 0;
    METRIC_TYPE_COUNTER        = 1;
    METRIC_TYPE_GAUGE          = 2;
    METRIC_TYPE_HISTOGRAM      = 3;
    METRIC_TYPE_GAUGEHISTOGRAM = 4;
    METRIC_TYPE_SUMMARY        = 5;
    METRIC_TYPE_INFO           = 6;
    METRIC_TYPE_STATESET       = 7;
  }
  MetricType type = 1;
  // help_ref is a reference to the Request.symbols array representing help
  // text for the metric. Help is optional, reference should point to an empty string in
  // such a case.
  uint32 help_ref = 3;
  // unit_ref is a reference to the Request.symbols array representing a unit
  // for the metric. Unit is optional, reference should point to an empty string in
  // such a case.
  uint32 unit_ref = 4;
}

// A native histogram message, supporting
// * sparse exponential bucketing, custom bucketing.
// * float or integer histograms.
//
// See the full spec: https://prometheus.io/docs/specs/native_histograms/
message Histogram { ... }
```

All timestamps MUST be int64 counted as milliseconds since the Unix epoch. Sample's values MUST be float64.

For every `TimeSeries` message:

- `labels_refs` MUST be provided.
- At least one element in `samples`, in `histograms`, or in `exemplars` MUST be provided (in `stateless` and `stateless-st` modes, at least one element in `samples` or in `histograms` MUST be provided). For example, a `TimeSeries` with only `metadata` and `labels_refs` is not valid.
- A `TimeSeries` MUST NOT include both `samples` and `histograms`. For series which (rarely) would mix float and histogram samples, a separate `TimeSeries` message MUST be used.
- `metadata` sub-fields SHOULD be provided (in `stateless` and `stateless-st` modes, `Metadata.type` MUST be specified). Receivers MAY reject series with unspecified `Metadata.type`.
- Exemplars SHOULD be provided if they exist for a series.

The following subsections define some schema elements in detail.

Symbols

The `io.prometheus.write.v2.Request` Protobuf Message is designed to intern all strings for the proven additional compression and memory efficiency gains on top of the standard compressions.

The `symbols` table MUST be provided, and it MUST contain deduplicated strings used in series, exemplar labels, and metadata strings. The first element of the `symbols` table MUST be an empty string, which is used to represent empty or unspecified values such as when `Metadata.unit_ref` or `Metadata.help_ref` are not provided. References MUST point to the existing index in the `symbols` string array.

Series Labels

The complete set of labels MUST be sent with each `Sample` or `Histogram` sample. Additionally, 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.

Metric names, label names, and label values MUST be any sequence of UTF-8 characters.

Metric names SHOULD adhere to the regex `[a-zA-Z_:]([a-zA-Z0-9_:])*`.

Label names SHOULD adhere to the regex `[a-zA-Z_]([a-zA-Z0-9_])*`.

Names that do not adhere to the above, might be harder to use for PromQL users (see the UTF-8 proposal for more details).

Label names beginning with "__" are RESERVED for system usage and SHOULD NOT be used, see Prometheus Data Model.

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

Samples and Histogram Samples

Senders MUST send `samples` (or `histograms`) for any given `TimeSeries` in timestamp order. Senders MAY send multiple requests for different series in parallel. In `stateless` and `stateless-st` modes, histogram metrics MUST be sent using `histograms` (e.g. Native Histograms with Custom Buckets) rather than classic histogram float `samples`.

Sample's or Histogram's `start_timestamp` SHOULD be provided for types that follow counter semantics (e.g. counters and counter histograms; in `stateless-st` mode, `start_timestamp` MUST be set for these types, except on stale markers). Receivers MAY reject those series without `start_timestamp` being set. Given optionality, the 0 value MUST be treated by receivers as unset value. To represent the unlikely 0 unix timestamp in milliseconds, "1" or "-1" value MUST be used.

Senders SHOULD send stale markers when a time series will no longer be appended to. Senders MUST send stale markers if the discontinuation of time series is possible to detect, for example:

- For series that were pulled (scraped), unless explicit timestamp was used.
- For series that is resulted by a recording rule evaluation.

Generally, not sending stale markers for series that are discontinued can lead to the Receiver non-trivial query time alignment issues.

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

Typically, Senders can detect when a time series will no longer be appended 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.

5. Tracking discontinuation of metrics for non-scrape source of metric (e.g. in k6 when the benchmark has finished for series per benchmark, it could emit a stale marker).

Metadata

Metadata SHOULD follow the official Prometheus guidelines for Type and Help.

Metadata MAY follow the official OpenMetrics guidelines for Unit.

Exemplars

Each exemplar, if attached to a `TimeSeries`:

- MUST contain a value.

- MAY contain labels e.g. referencing trace or request ID. If the exemplar references a trace it SHOULD use the `trace_id` label name, as a best practice.
- MUST contain a timestamp. While exemplar timestamps are optional in Prometheus/Open Metrics exposition formats, the assumption is that a timestamp is assigned at scrape time in the same way a timestamp is assigned to the scrape sample. Receivers require exemplar timestamps to reliably handle (e.g. deduplicate) incoming exemplars.
- MAY be sent in a `TimeSeries` that carries neither samples nor histograms (except in `stateless` and `stateless-st` modes, where exemplars MUST only be sent on a `TimeSeries` that also contains at least one element in `samples` or in `histograms`). For cases where there are multiple `samples` or `histograms` on a `TimeSeries`, all exemplars MUST be associated with all of those `samples` or `histograms`.

Modes

By default (`none` mode, when no `;mode=` parameter is set or when `;mode=none` is specified), `io.prometheus.write.v2.Request` keeps several structural and metadata rules optional (`SHOULD` or `MAY`) so Senders can forward any scraped metric without restrictions or conversion overhead. Senders MAY omit `;mode=` or explicitly send `;mode=none` to indicate this baseline mode. However, some common stateless or strongly typed Receivers cannot ingest all the data e.g. detached exemplars, classic histogram series, untyped metrics, or counters without start timestamps.

To address this, `io.prometheus.write.v2.Request` defines two progressive modes, `stateless` and `stateless-st`, that tighten optional rules into `MUST` requirements on Senders while keeping every payload a valid `io.prometheus.write.v2.Request` message. See the FAQ for the rationale.

Senders SHOULD support `stateless` and `stateless-st` modes. Senders MAY operate in either mode behind configuration or by default.

Receivers MAY accept any valid `io.prometheus.write.v2.Request` regardless of mode, require at least `stateless` mode, or require `stateless-st` mode only. If rejected, 415 HTTP Unsupported

Media Type MUST be used (see Unsupported Request Content).

`stateless` Mode

When operating in `stateless` mode, Senders MUST:

- Use content type of `Content-Type: application/x-protobuf;proto=io.prometheus.write.v2.Request;mode=stateless`.
- Ensure all of the following for every `TimeSeries` in a `Request`:
  - `Exemplar`s MUST only be sent on a `TimeSeries` that also contains at least one element in `samples` or in `histograms`. Exemplar-only `TimeSeries` messages MUST NOT be sent.
  - Histograms MUST use the `Histogram` message (e.g. using Native Histograms with Custom Buckets instead of classic histogram `_bucket`, `_sum`, and `_count` float `Sample` series).
  - `Metadata.type` MUST be specified and MUST NOT be `METRIC_TYPE_UNSPECIFIED` (`0`). For series with an unknown metric type (such as `untyped` scraped metrics or recording rule outputs), Senders SHOULD assume:
    - `METRIC_TYPE_GAUGE`, unless the metric name has a `_total` suffix, in which case Senders SHOULD assume `METRIC_TYPE_COUNTER`.
    - `Metadata.help_ref` set to `"(type inferred from untyped)"` if empty, or with `" (type inferred from untyped)"` appended to the existing help text.

`stateless-st` Mode

When operating in `stateless-st` mode, Senders MUST:

- Use content type of `Content-Type: application/x-protobuf;proto=io.prometheus.write.v2.Request;mode=stateless-st`.
- Satisfy all `TimeSeries` requirements of `stateless` mode.
- Ensure that `start_timestamp` is set (non-zero) on every `Sample` and `Histogram` (except stale markers) for metric types `METRIC_TYPE_COUNTER`, `METRIC_TYPE_HISTOGRAM`, as well as `METRIC_TYPE_SUMMARY` `_sum` and `_count` `Samples`.

Out of Scope

The same as in 1.0.

Future Plans

This section contains speculative plans that are not considered part of protocol specification yet but are mentioned here for completeness. Note that 2.0 specification completed 2 of 3 future plans in the 1.0.

- **Transactionality** There is still no transactionality defined for 2.0 specification, mostly because it makes a scalable Sender implementation difficult. Prometheus Sender 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.
  However, Remote-Write 2.0 specification solves an important transactionality problem for the classic histogram buckets. This is done thanks to the native histograms supporting custom bucketing (NHCB) in the `io.prometheus.write.v2.Request` wire format, which `stateless` and `stateless-st` modes mandate instead of classic histograms. When no mode is specified, Senders might still translate classic histograms to native histograms this way, and Receivers MAY ignore certain metric types (e.g. classic histograms).
- **Alternative wire formats**. The OpenTelemetry community has shown the validity of Apache Arrow (and potentially other columnar formats) for over-wire data transfer with their OTLP protocol. We would like to do experiments to confirm the compatibility of a similar format with Prometheus' data model and include benchmarks of any resource usage changes. We would potentially maintain both a protobuf and columnar format long term for compatibility reasons and use our content negotiation to add different Protobuf Messages for this purpose.
- **Global symbols**. Pre-defined string dictionary for interning The protocol could pre-define a static dictionary of ref->symbol that includes strings that are considered common, e.g. "namespace", "le", "job", "seconds", "bytes", etc. Senders could refer to these without the need to include them in the request's symbols table. This dictionary could incrementally grow with minor version releases of this protocol.

Related

FAQ

Why did you not use gRPC? Because the 1.0 protocol does not use gRPC, breaking it would increase friction in the adoption. See 1.0 reason.

Why not stream 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 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 optimized for append-only workloads. However, this requirement is also shared across many other databases and vendors in the ecosystem. In fact, Prometheus with OOO feature enabled, allows out-of-order writes, but with the performance penalty, thus reserved for rare events. To sum up, Receivers may support out-of-order write, though it is not permitted by the specification. In the future e.g. 2.x spec versions, we could extend content type to negotiate the out-of-order writes, if needed.

What's the purpose of `stateless` and `stateless-st` modes? By default (`none` mode, when no mode or `;mode=none` is specified), Remote-Write 2.0 allows any valid permutation of fields in `io.prometheus.write.v2.Request` to preserve compatibility and minimal overhead for existing Prometheus TSDB pipelines.

However, stateless or strongly-typed receivers often require structured histograms, attached exemplars, metric types, or start timestamps to ingest metrics without stateful caches or mid-stream `400 HTTP Bad Request` rejections. Modes have two goals:

- Simplify communication between senders and receivers, such as configuring required sender features as a single setting, rejecting unsupported modes early on the receiver with `415 HTTP Unsupported Media Type`, and documenting the cost and requirements of each mode.
- Help the ecosystem adopt stricter semantics, such as letting receivers advertise which mode they require and letting operators monitor mode adoption across senders.

We defined two progressive modes, `stateless` and `stateless-st`, rather than a single mode or fine-grained capability flags:

- `stateless` mode rules (attaching exemplars to samples or histograms, converting classic histograms to NHCB, and requiring `Metadata.type` with a `METRIC_TYPE_GAUGE` or `_total`-based `METRIC_TYPE_COUNTER` fallback for unknown metric types) make every `TimeSeries` message self-contained and typed. Any sender can enforce `stateless` mode without dropping data or tracking cross-scrape counter start timestamps, and receivers can validate histogram structure because `Metadata.type` is always set.
- `stateless-st` mode adds the `start_timestamp` requirement on cumulative metric types (`METRIC_TYPE_COUNTER`, `METRIC_TYPE_HISTOGRAM`, and summary `_sum` and `_count`). It is kept separate from `stateless` because `start_timestamp` depends on upstream instrumentation exposing created timestamps or stateful sender-side synthesis, and adds per-sample wire overhead on cumulative series that not all receivers require.
- `stateless-st` includes `stateless` because receivers that require `start_timestamp` on cumulative metrics also require `Metadata.type`, structured histograms, and attached exemplars, and validating `start_timestamp` itself requires knowing `Metadata.type`.

  Following RFC 9413, three progressive levels (`none`, `stateless`, `stateless-st`) avoid `2^N` protocol permutations, keep `Content-Type` matching to a single parameter value, and keep compliance testing linear.

In future specification versions, we might propose making `stateless` mode the default when `;mode=` is omitted while keeping `none` and `stateless-st` explicitly selectable, depending on ecosystem feedback and adoption.

How can we parallelise requests with the in-order constraint? Samples must be in-order *for a given series*. However, even if a Receiver does not support out-of-order write, the Remote-Write requests can be sent in parallel as long as they are for different series. Prometheus shards 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.

What are the differences between Remote-Write 2.0 and OpenTelemetry's OTLP protocol? OpenTelemetry OTLP is a protocol for transporting of telemetry data (such as metrics, logs, traces and profiles) between telemetry sources, intermediate nodes and telemetry backends. The recommended transport involves gRPC with protobuf, but HTTP with protobuf or JSON are also described. It was designed from scratch with the intent to support a variety of different observability signals, data types and extra information. For metrics that means additional non-identifying labels, flags, temporal aggregations types, resource or scoped metrics, schema URLs and more. OTLP also requires the semantic convention to be used.

Remote-Write was designed for simplicity, efficiency and organic growth. The first version was officially released in 2023, when already dozens of battle-tested adopters in the CNCF ecosystem had been using this protocol for years. Remote-Write 2.0 iterates on the previous protocol by adding a few new elements (metadata, exemplars, start timestamp and native histograms) and string interning. Remote-Write 2.0 is always stateless, focuses only on metrics and is opinionated; as such it is scoped down to elements that the Prometheus community considers enough to have a robust metric solution. The intention is to ensure the Remote-Write is a stable protocol that is cheaper and simpler to adopt and use than the alternatives in the observability ecosystem.

□□□ #2
□ □□□ □□□ 2026-10-31 13:21:59 UTC
□ □□□ □□□ 2026-10-31 13:22:01 UTC