Go gRPC Interceptors for Prometheus monitoring

build error go report A+ GoDoc Reference ✱ used by 1.2k projects codecov 72%

License Apache 2.0

Prometheus monitoring for your gRPC Go servers and clients.

A sister implementation for gRPC Java (same metrics, same semantics) is in grpc-ecosystem/java-grpc-prometheus.

Interceptors

gRPC Go recently acquired support for Interceptors, i.e. middleware that is executed by a gRPC Server before the request is passed onto the user's application logic. It is a perfect way to implement common patterns: auth, logging and... monitoring.

To use Interceptors in chains, please see `go-grpc-middleware` .

Usage

There are two types of interceptors: client-side and server-side. This package provides monitoring Interceptors for both.

Server-side

```
import "github.com/grpc-ecosystem/go-grpc-prometheus"
...
    // Initialize your gRPC server's interceptor.
    myServer := grpc.NewServer(
        grpc.StreamInterceptor(grpc_prometheus.StreamServerInterceptor),
        grpc.UnaryInterceptor(grpc_prometheus.UnaryServerInterceptor),
    )
    // Register your gRPC service implementations.
    myservice.RegisterMyServiceServer(s.server, &myServiceImpl{})
    // After all your registrations, make sure all of the Prometheus metrics are
initialized.
    grpc_prometheus.Register(myServer)
    // Register Prometheus metrics handler.
    http.Handle("/metrics", promhttp.Handler())
...
```

Client-side

```
import "github.com/grpc-ecosystem/go-grpc-prometheus"
...
   clientConn, err = grpc.Dial(
       address,
           grpc.WithUnaryInterceptor(grpc_prometheus.UnaryClientInterceptor),
           grpc.WithStreamInterceptor(grpc_prometheus.StreamClientInterceptor)
   )
   client = pb_testproto.NewTestServiceClient(clientConn)
   resp, err := client.PingEmpty(s.ctx, &myservice.Request{Msg: "hello"})
...
```

Metrics

Labels

All server-side metrics start with `grpc_server` as Prometheus subsystem name. All client-side metrics start with `grpc_client`. Both of them have mirror-concepts. Similarly all methods contain the same rich labels:

- `grpc_service` - the gRPC service name, which is the combination of protobuf `package` and the `grpc_service` section name. E.g. for `package = mwitkow.testproto` and `service TestService` the label will be `grpc_service="mwitkow.testproto.TestService"`

- `grpc_method` - the name of the method called on the gRPC service. E.g. `grpc_method="Ping"`

- `grpc_type` - the gRPC type of request. Differentiating between the two is important especially for latency measurements.

    - `unary` is single request, single response RPC
    - `client_stream` is a multi-request, single response RPC
    - `server_stream` is a single request, multi-response RPC
    - `bidi_stream` is a multi-request, multi-response RPC

Additionally for completed RPCs, the following labels are used:

- `grpc_code` - the human-readable gRPC status code. The list of all statuses is to long, but here are some common ones:

    - `OK` - means the RPC was successful
    - `IllegalArgument` - RPC contained bad values
    - `Internal` - server-side error not disclosed to the clients

Counters

The counters and their up to date documentation is in server_reporter.go and client_reporter.go the respective Prometheus handler (usually `/metrics`).

For the purpose of this documentation we will only discuss `grpc_server` metrics. The `grpc_client` ones contain mirror concepts.

For simplicity, let's assume we're tracking a single server-side RPC call of `mwitkow.testproto.TestService`, calling the method `PingList`. The call succeeds and returns 20 messages in the stream.

First, immediately after the server receives the call it will increment the `grpc_server_started_total` and start the handling time clock (if histograms are enabled).

```
grpc_server_started_total{grpc_method="PingList",grpc_service="mwitkow.testproto.TestSe
 1
```

Then the user logic gets invoked. It receives one message from the client containing the request (it's a `server_stream`):

```
grpc_server_msg_received_total{grpc_method="PingList",grpc_service="mwitkow.testproto.
 1
```

The user logic may return an error, or send multiple messages back to the client. In this case, on each of the 20 messages sent back, a counter will be incremented:

```
grpc_server_msg_sent_total{grpc_method="PingList",grpc_service="mwitkow.testproto.Test
 20
```

After the call completes, its status (`OK` or other gRPC status code) and the relevant call labels increment the `grpc_server_handled_total` counter.

```
grpc_server_handled_total{grpc_code="OK",grpc_method="PingList",grpc_service="mwitkow.
 1
```

Histograms

Prometheus histograms are a great way to measure latency distributions of your RPCs. However, since it is bad practice to have metrics of high cardinality the latency monitoring metrics are disabled by default. To enable them please call the following in your server initialization code:

```
grpc_prometheus.EnableHandlingTimeHistogram()
```

After the call completes, its handling time will be recorded in a Prometheus histogram variable `grpc_server_handling_seconds`. The histogram variable contains three sub-metrics:

- `grpc_server_handling_seconds_count` - the count of all completed RPCs by status and method
- `grpc_server_handling_seconds_sum` - cumulative time of RPCs by status and method, useful for calculating average handling times
- `grpc_server_handling_seconds_bucket` - contains the counts of RPCs by status and method in respective handling-time buckets. These buckets can be used by Prometheus to estimate SLAs (see here)

The counter values will look as follows:

```
grpc_server_handling_seconds_bucket{grpc_code="OK",grpc_method="PingList",grpc_service=
 1
grpc_server_handling_seconds_bucket{grpc_code="OK",grpc_method="PingList",grpc_service=
 1
grpc_server_handling_seconds_bucket{grpc_code="OK",grpc_method="PingList",grpc_service=
 1
grpc_server_handling_seconds_bucket{grpc_code="OK",grpc_method="PingList",grpc_service=
 1
```

```
grpc_server_handling_seconds_bucket{grpc_code="OK",grpc_method="PingList",grpc_service=
 1
grpc_server_handling_seconds_bucket{grpc_code="OK",grpc_method="PingList",grpc_service=
 1
grpc_server_handling_seconds_bucket{grpc_code="OK",grpc_method="PingList",grpc_service=
 1
grpc_server_handling_seconds_bucket{grpc_code="OK",grpc_method="PingList",grpc_service=
 1
grpc_server_handling_seconds_bucket{grpc_code="OK",grpc_method="PingList",grpc_service=
 1
grpc_server_handling_seconds_bucket{grpc_code="OK",grpc_method="PingList",grpc_service=
 1
grpc_server_handling_seconds_bucket{grpc_code="OK",grpc_method="PingList",grpc_service=
 1
grpc_server_handling_seconds_bucket{grpc_code="OK",grpc_method="PingList",grpc_service=
 1
grpc_server_handling_seconds_sum{grpc_code="OK",grpc_method="PingList",grpc_service="m
 0.0003866430000000001
grpc_server_handling_seconds_count{grpc_code="OK",grpc_method="PingList",grpc_service="
 1
```

Useful query examples

Prometheus philosophy is to provide raw metrics to the monitoring system, and let the aggregations be handled there. The verbosity of above metrics make it possible to have that flexibility. Here's a couple of useful monitoring queries:

request inbound rate

```
sum(rate(grpc_server_started_total{job="foo"}[1m])) by (grpc_service)
```

For `job="foo"` (common label to differentiate between Prometheus monitoring targets), calculate the rate of requests per second (1 minute window) for each gRPC `grpc_service` that the job has. Please note how the `grpc_method` is being omitted here: all methods of a given gRPC service will be summed together.

unary request error rate

```
sum(rate(grpc_server_handled_total{job="foo",grpc_type="unary",grpc_code!="OK"}
[1m])) by (grpc_service)
```

For `job="foo"`, calculate the per-`grpc_service` rate of `unary` (1:1) RPCs that failed, i.e. the ones that didn't finish with `OK` code.

unary request error percentage

```
sum(rate(grpc_server_handled_total{job="foo",grpc_type="unary",grpc_code!="OK"}
[1m])) by (grpc_service)
 /
```

```
sum(rate(grpc_server_started_total{job="foo",grpc_type="unary"}[1m])) by
(grpc_service)
 * 100.0
```

For `job="foo"`, calculate the percentage of failed requests by service. It's easy to notice that this is a combination of the two above examples. This is an example of a query you would like to alert on in your system for SLA violations, e.g. "no more than 1% requests should fail".

average response stream size

```
sum(rate(grpc_server_msg_sent_total{job="foo",grpc_type="server_stream"}[10m])) by
(grpc_service)
 /
sum(rate(grpc_server_started_total{job="foo",grpc_type="server_stream"}[10m])) by
(grpc_service)
```

For `job="foo"` what is the `grpc_service`-wide `10m` average of messages returned for all `server_stream` RPCs. This allows you to track the stream sizes returned by your system, e.g. allows you to track when clients started to send "wide" queries that ret Note the divisor is the number of started RPCs, in order to account for in-flight requests.

99%-tile latency of unary requests

```
histogram_quantile(0.99,
  sum(rate(grpc_server_handling_seconds_bucket{job="foo",grpc_type="unary"}[5m])) by
(grpc_service,le)
)
```

For `job="foo"`, returns an 99%-tile quantile estimation of the handling time of RPCs per service. Please note the `5m` rate, this means that the quantile estimation will take samples in a rolling `5m` window. When combined with other quantiles (e.g. 50%, 90%), this query gives you tremendous insight into the responsiveness of your system (e.g. impact of caching).

percentage of slow unary queries (>250ms)

```
100.0 - (
sum(rate(grpc_server_handling_seconds_bucket{job="foo",grpc_type="unary",le="0.25"}
[5m])) by (grpc_service)
 /
sum(rate(grpc_server_handling_seconds_count{job="foo",grpc_type="unary"}[5m])) by
(grpc_service)
) * 100.0
```

For `job="foo"` calculate the by-`grpc_service` fraction of slow requests that took longer than `0.25` seconds. This query is relatively complex, since the Prometheus aggregations use `le` (less or equal) buckets, meaning that counting "fast" requests fractions is easier. However, simple maths helps. This is an example of a query you would like to alert on in your system for SLA violations, e.g. "less than 1% of requests are slower than 250ms".

Status

This code has been used since August 2015 as the basis for monitoring of *production* gRPC micro services at Improbable.

License

`go-grpc-prometheus` is released under the Apache 2.0 license. See the LICENSE file for details.