Prometheus High Availability (2): Understanding Remote Storage

2018-03-07 | Prometheus | 0 Comments

Prometheus' local storage design reduces the complexity of its own operations and management while meeting the needs of most user monitoring scales. But local storage also means that Prometheus can't persist data, can't store large amounts of historical data, and can't scale flexibly.

In order to keep Prometheus simple, Prometheus did not try to solve the above problem in itself, but by defining two standard interfaces (remote_write/remote_read), users can dock any third-party storage service based on these two interfaces. The way to become Remote Storage in Promthues.

Remote Write

The user can specify the URL address of the Remote Write in the Promtheus configuration file. Once the configuration item is set, Prometheus sends the sample data to the adapter (Adaptor) via HTTP. The user can dock any external service in the adapter. The external service can be a real storage system, a public cloud storage service, or any form of message queue.

Remote Write

Remote Read

As shown in the figure below, Promthues' Remote Read is also implemented by an adapter. In the remote read process, when the user initiates the query request, Promthues will initiate a query request (matchers, ranges) to the URL configured in remote_read, and Adaptor obtains the response data from the third-party storage service according to the request condition. At the same time, the original sample data of the data converted to Promthues is returned to Prometheus Server.

After obtaining the sample data, Promthues uses PromQL to reprocess the sample data locally.

> Note: Even with remote reads, the processing of rule files in Prometheus and the processing of the Metadata API are done only locally.

Remote Read

Configuration file

When users need to use the remote read and write function, mainly by adding the remote_write and remote_read configurations in the Prometheus configuration file, where url is used to specify the HTTP address of the remote read/write. If the URL is authenticated, the security authentication configuration can be performed via basic_auth. For

https support you need to set tls_concig. Proxy_url is mainly used when Prometheus cannot directly access the adapter service.

The specific configuration of remote_write and remote_write is as follows:

```
remote_write:
    url: <string>
    [ remote_timeout: <duration> | default = 30s ]
    write_relabel_configs:
    [ - <relabel_config> ... ]
    basic_auth:
    [ username: <string> ]
    [ password: <string> ]
    [ bearer_token: <string> ]
    [ bearer_token_file: /path/to/bearer/token/file ]
    tls_config:
    [ <tls_config> ]
    [ proxy_url: <string> ]

remote_read:
    url: <string>
    required_matchers:
    [ <labelname>: <labelvalue> ... ]
    [ remote_timeout: <duration> | default = 30s ]
    [ read_recent: <boolean> | default = false ]
    basic_auth:
    [ username: <string> ]
    [ password: <string> ]
    [ bearer_token: <string> ]
    [ bearer_token_file: /path/to/bearer/token/file ]
    [ <tls_config> ]
    [ proxy_url: <string> ]
```

Customize the Remote Stoarge Adaptor

Implementing a custom Remote Storage requires the user to create separate HTTP services to support remote_read and remote_write.

Remote Storage

The protocols related to Remote Storage in Prometheus are currently defined by the following proto files:

```
syntax = "proto3";
package prometheus;

```

```
option go_package = "prompb";

import "types.proto";

message WriteRequest {
  repeated prometheus.TimeSeries timeseries = 1;
}

message ReadRequest {
  repeated Query queries = 1;
}

message ReadResponse {
  // In same order as the request's queries.
  repeated QueryResult results = 1;
}

message Query {
  int64 start_timestamp_ms = 1;
  int64 end_timestamp_ms = 2;
  repeated prometheus.LabelMatcher matchers = 3;
}

message QueryResult {
  // Samples within a time series must be ordered by time.
  repeated prometheus.TimeSeries timeseries = 1;
}
```

The following code shows a simple remote_write service that creates an HTTP service for receiving remote_write. After converting the request content into a WriteRequest, the user can perform subsequent logical processing according to their needs.

```
package main

import (
        "fmt"
        "io/ioutil"
        "net/http"

        "github.com/gogo/protobuf/proto"
        "github.com/golang/snappy"
        "github.com/prometheus/common/model"

        "github.com/prometheus/prometheus/prompb"
)

```

```
15  func main() {
16          http.HandleFunc("/receive", func(w http.ResponseWriter, r *http.Request) {
17                  compressed, err := ioutil.ReadAll(r.Body)
18                  if err != nil {
19                          http.Error(w, err.Error(), http.StatusInternalServerError)
20                          return
21                  }
22
23                  reqBuf, err := snappy.Decode(nil, compressed)
24                  if err != nil {
25                          http.Error(w, err.Error(), http.StatusBadRequest)
26                          return
27                  }
28
29                  var req prompb.WriteRequest
30                  if err := proto.Unmarshal(reqBuf, &req); err != nil {
31                          http.Error(w, err.Error(), http.StatusBadRequest)
32                          return
33                  }
34
35                  for _, ts := range req.Timeseries {
36                          m := make(model.Metric, len(ts.Labels))
37                          for _, l := range ts.Labels {
38                                  m[model.LabelName(l.Name)] = model.LabelValue(l.Value)
39                          }
40                          fmt.Println(m)
41
42                          for _, s := range ts.Samples {
43                                  fmt.Printf("  %f %d\n", s.Value, s.Timestamp)
44                          }
45                  }
46          })
47
48          http.ListenAndServe(":1234", nil)
49  }
```

Use Influxdb as a Remote Stoarge

The Prometheus community also offers some Remote Storage support for third-party databases:

Storage service	Support mode
AppOptics	write

Storage service	Support mode
Chronix	write
Cortex:	read/write
CrateDB	read/write
Gnocchi	write
Graphite	write
InfluxDB	read/write
OpenTSDB	write
PostgreSQL/TimescaleDB:	read/write
SignalFx	write

Here we demonstrate how Influxdb will be used as Prometheus' Remote Storage to ensure that historical data can be recovered and retrieved from Influxdb when Prometheus crashes or restarts.

Here we use docker-compose to define and start the Influxdb database service. docker-compose.yml is defined as follows:

```
version: '2'
services:
  influxdb:
    image: influxdb:1.3.5
    command: -config /etc/influxdb/influxdb.conf
    ports:
      - "8086:8086"
    environment:
      - INFLUXDB_DB=prometheus
      - INFLUXDB_ADMIN_ENABLED=true
      - INFLUXDB_ADMIN_USER=admin
      - INFLUXDB_ADMIN_PASSWORD=admin
      - INFLUXDB_USER=prom
      - INFLUXDB_USER_PASSWORD=prom
```

Start the influxdb service

```
$ docker-compose up -d
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STA
795d0ead87a1        influxdb:1.3.5      "/entrypoint.sh -c..."   3 hours ago         Up
```

Obtain and launch the Remote Storage Adapter provided by Prometheus:

```
go get github.com/prometheus/prometheus/documentation/examples/remote_storage/remote_sto
```

After getting the remote_storage_adapter source, go will automatically compile the relevant source into an executable file and save it in the $GOPATH/bin/ directory.

Start remote_storage_adapter and set the Influxdb related authentication information:

```
INFLUXDB_PW=prom $GOPATH/bin/remote_storage_adapter -influxdb-url=http://localhost:8086
```

Modify prometheus.yml to add Remote Storage related configuration content:

```
remote_write:
  - url: "http://localhost:9201/write"

remote_read:
  - url: "http://localhost:9201/read"
```

After restarting Prometheus to get the data, log in to the influxdb container and verify the data write. As shown below, Promtheus related metrics can be seen when the data can be written to Influxdb normally.

```
docker exec -it 795d0ead87a1 influx
Connected to http://localhost:8086 version 1.3.5
InfluxDB shell version: 1.3.5
> auth
username: prom
password:

> use prometheus
> SHOW MEASUREMENTS
name:  measurements
name
- ---
go_gc_duration_seconds
```

```
go_gc_duration_seconds_count
go_gc_duration_seconds_sum
go_goroutines
go_info
go_memstats_alloc_bytes
go_memstats_alloc_bytes_total
go_memstats_buck_hash_sys_bytes
go_memstats_frees_total
go_memstats_gc_cpu_fraction
go_memstats_gc_sys_bytes
go_memstats_heap_alloc_bytes
go_memstats_heap_idle_bytes
```

When the data is successfully written, stop the Prometheus service. Also delete the Prometheus data directory, simulate Promthues data loss and restart Prometheus. Open the Prometheus UI. If the configuration is normal, Prometheus can normally query the historical data records deleted by the local storage.

Get historical data from Remote Storage

Next

The Remote Storage feature can be used to store the monitoring sample data in Promthues in a third-party storage service, thus solving the data persistence problem of Promthues. At the same time, due to the restriction of local storage, Promthues itself can also be flexibly extended, and dynamic scheduling can be performed in an environment such as Kubernetes.

After solving the problem of data persistence and resilience, the author will introduce another feature of the Promthues federation cluster, which can be used to implement the horizontal expansion and functional partitioning of Promthues.

Prometheus High Availability (1): Understanding Local Storage

Prometheus High Availability (3): Federal Cluster