

503錯誤處理方法



使用的是 K8s

vminsert有掛VirtualService、Service，vmalert的-remoteWrite.url是vminsert的VirtualService

這是出現在vmalert寫入alert到vminsert時出現的log，503，可是vminsert 有把alert寫進vmstorage

vmalert log出現這個，我找不到處理方法：

2025-08-01T16:56:03.749+0800

warn

VictoriaMetrics/app/ vmalert/remotewrite/client.go:256

attempt 1 to send

request failed: unexpected response code 503 for http://vm-vminsert.ifond.edg.f18.tsmc

.com/insert/0/prometheus/api/v1 /write. Response body "upstream connect error or disconnect/reset before headers. reset reason: connection termination"

(retriable: true)2025-08-01



已思考 1m 2s

- Istio 通常只有在進行 ALPN 協商時才會使用 HTTP/2，但 Pod 之間的連接可能會強制使用 mTLS 並依舊使用 HTTP/2。VictoriaMetrics 支援 HTTP/1.1，似乎也能處理 HTTP/2。如果 Envoy 終止 mTLS 並代理 HTTP/1.1，應該不會有問題，但錯誤提示表明 Envoy 可能無法建立或維持連線。這可能與 upstream 的閒置超時或 Pod 重啟等因素有關。
- vmalert 配置成推送警報規則的時間序列到 /write，若 vminsert 將資料寫入 vmstorage 時即使遇到 503 錯誤，最終可能會因重試成功。確認 vmalert 的 remoteWrite.url 是否指向穩定的 vminsert 集群 DNS，並建議使用 K8s ClusterIP 服務而非 VirtualService 主機。
- 在 VirtualService 中設置重新嘗試和超時很重要，例如將重試次數設為 3 次，每次嘗試時間限制為 30 秒，並設置對 5xx 錯誤等進行重試。此外，檢



查 vmalet 的 remoteWrite 設定，特別是傳輸大小和超時設置，可能需要進一步調整。

- 檢查 vminsert 的 Pod 中斷預算，確保 HPA 沒有終止 Pod。此外，對於 Istio sidecar 資源，設定 `proxy.istio.io/config` 來延長 `holdApplicationUntilProxyStarts` 和 `terminationDrainDuration`，並確定 livenessProbe 有啟用 grpc 健康檢查。檢查 API 路徑並確認端口與協議。
- 檢查連線問題，確認是否為 Envoy 端的 503 錯誤，而且了解是否與 mTLS、健康檢查或網路超時有關。建議繞過 Istio 使用 ClusterIP 服務，或配置正確的 VirtualService 及 DestinationRule 來解決過多重試、超時或連線池相關問題。

☑ 完成

先幫你拆解這個 503 訊息的意義，再給一套「確認 → 立即止血 → 穩定化」的做法與可直接套用的 YAML。

發生了什麼？

```
`upstream connect error or disconnect/reset before headers. reset reason: connection termination`
```

這是 **Envoy/Istio** 在還沒拿到上游（vminsert）的 **HTTP header** 前就被關連線的典型訊息。常見原因：

- vminsert Pod 短暫重啟 / OOM / readiness 轉不過來 → 連線被關。
- 透過 **VirtualService** 走 sidecar 時的 **mTLS/協定 (HTTP/1.1 vs h2c) /timeout/連線池** 設定不合或太保守。
- 連線空閒被回收（idle timeout / keepalive 不對齊）或 rolling restart/drain 期切到關閉中的 Pod。
- URL 或 DNS 解析偶發問題（含 **URL 末端的空白字元** 等）。

“為什麼資料仍寫進 vmstorage？”

vmalet 會重試 (retriable: true)，所以有些請求失敗、但後續重試成功，就會看到「有寫進去，但仍有 503 warn」。

先做 6 項快速檢查（5 分鐘可抓到 80% 問題）

1. 確認 URL 是否乾淨

你的 log 斷行看起來像：``.../api/v1 /write``（中間多了空白）。請確定 ``-remoteWrite.url`` 沒有多餘空白/換行。

小技巧（在 vmaalert Pod 內）：

```
echo -n "$REMOTEWRITE_URL" | hexdump -C
```

結尾若不是 ``... 2f 77 72 69 74 65``（/write）就要清掉奇怪字元。

2. 就近用 ClusterIP 直連測試（先暫避 VirtualService）

從 vmaalert Pod 內做：

```
nc -vz vminsert.<ns>.svc.cluster.local 8480
curl -sS -v http://vminsert.<ns>.svc.cluster.local:8480/health
```

- 成功：表示 vminsert 活著，先把 ``-remoteWrite.url`` 改走 ClusterIP（止血法，見下）。
- 失敗：優先排查 vminsert 的 Pod/Service/Endpoints。

3. 看 vminsert 是否在重啟或被 OOMKill

```
kubectl -n <ns> get pod -l app=vminsert -o wide
kubectl -n <ns> describe pod -l app=vminsert | egrep -i "Restart|OOM"
```

4. 確定 Service port 命名正確

- Istio 需靠 **port name** 判斷協定。vminsert HTTP 入口的 Service port 請命名成 ``http``（例如 ``name: http-8480``）。命名成 ``tcp-*`` 會讓 VS 規則吃不到或走錯 filter chain。

5. 檢查 Istio 端點視圖（看是否有壞 endpoint 或連線失敗）

```
istioctl pc endpoints <vmalert-pod> -n <ns> --cluster
"outbound|8480||vminsert.<ns>.svc.cluster.local"
istioctl pc clusters <vmalert-pod> -n <ns> | grep vminsert
istioctl pc stats <vmalert-pod> -n <ns> --pattern ".*vminsert.*
(upstream_cx_connect_fail|upstream_rq_5xx|upstream_cx_destroy_remote)"
```

6. mTLS 模式一致性

- vminsert 若 在 **mesh**：DestinationRule 要 `tls.mode: ISTIO_MUTUAL`。
- vminsert 若 不在 **mesh**：請 `tls.mode: DISABLE`（避免加密到一個不會講 mTLS 的端點）。

立即止血（最簡單、最穩定的路徑）

把 **vmalert** 直接打到 **vminsert** 的 **ClusterIP Service**，先繞過 VirtualService / Gateway 的不確定因素：

```
-remoteWrite.url=http://vminsert.
<ns>.svc.cluster.local:8480/insert/0/prometheus/api/v1/write
```

“兩個重點：”

1. “**Service 名稱 + .svc.cluster.local + port 8480**（依你的部署調整）”
2. “**確認 URL 沒有多餘空白**（真的很常中）”

這樣通常就會把 503 降到幾乎沒有。

若你必須經過 Istio (VirtualService/DestinationRule) — 推薦設定

“目標：放寬 *timeout*、避免 *h2* 升級問題、加強連線池與重試，並避免把流量打到正在熔斷/重啟中的 *Pod*。”

VirtualService (重試與超時)

```
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: vminsert
  namespace: <ns>
spec:
  hosts:
    - vm-vminsert.ifond.edg.f18.tsmc.com # 你的 FQDN
  http:
    - route:
        - destination:
            host: vminsert.<ns>.svc.cluster.local
            port:
                number: 8480
      retries:
        attempts: 3
        perTryTimeout: 10s
        retryOn: 5xx,connect-failure,refused-stream,reset,gateway-error,stream-
reset
      timeout: 30s
```

DestinationRule (連線池、outlier、TLS、禁升級 h2)

```

apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: vminsert
  namespace: <ns>
spec:
  host: vminsert.<ns>.svc.cluster.local
  trafficPolicy:
    # 若 vminsert 在 mesh : 用 ISTIO_MUTUAL ; 不在 mesh : 改 DISABLE
    tls:
      mode: ISTIO_MUTUAL
    connectionPool:
      tcp:
        maxConnections: 2000
        connectTimeout: 5s
        tcpKeepalive:
          time: 7200s
          interval: 75s
      http:
        # 避免升級到 HTTP/2 (部分後端/中間層對 h2c 行為不一致)
        h2UpgradePolicy: DO_NOT_UPGRADE
        http1MaxPendingRequests: 10000
        maxRequestsPerConnection: 0      # 0 = 不限制, 避免頻繁關連線
        idleTimeout: 120s                # 放寬空閒連線時間
    outlierDetection:
      consecutive5xx: 20
      interval: 5s
      baseEjectionTime: 30s
    loadBalancer:
      simple: ROUND_ROBIN

```

“另外：**Service port** 命名請用 `name: http-8480` (或至少 `http` 開頭)。”

vmalert 端可以再加的穩定化 (可選)



- **送出逾時**（例如 30s）與**佇列大小**（避免尖峰時丟包/大量重試）適度放寬。
（參數名稱依你使用的 vmaalert 版本，通常有 send timeout / queue / batch 類型設定。原則：稍微放寬，但別無限大）
- 若你看到大量 intermittent 503，但最終都有寫入，可先把 log level 維持預設，只要頻率不高就算「可接受抖動」。

故障排除決策樹（簡版）

1. **ClusterIP 直連 OK** → 問題多半在 Istio 路由/策略 → 套用上面 VS/DR。
2. **ClusterIP 直連也偶發失敗** → 看 vminsert Pod（重啟/OOM/CPU 飆高）、節點壓力、PDB/HPA。
3. **只在滾動更新或證書輪替時發生** → 增加 `terminationDrainDuration``、調整 PDB，並避免把流量打到 draining 中的 Pod。
4. **只在閒置後第一發失敗** → 提高 `idleTimeout``、啟用 `tcpKeepalive``、增大 `connectTimeout``。

小結

- 你的訊息是 **Envoy 在還沒拿到 header 前連線就被關**，與 vminsert 寫入成功不矛盾，因為重試後成功。
- 最快的止血是 **把 vmaalert 直接指到 vminsert 的 ClusterIP**（確認 URL 乾淨、port 名稱為 http）。
- 若一定要走 Istio，按上面的 **VS/DR 範本** 去放寬 timeout、禁 h2 升級、加大連線池與 outlier 設定，通常可以把這種 503 幾乎消掉。

如果你願意，把目前的 **VirtualService / DestinationRule / Service YAML**（以及 `remoteWrite.url`` 的實際值）貼上來，我可以直接幫你改成可用的版本。



```
這是VictoriaMetrics/app/ vmaalert/remotewrite/client.go
package remotewrite

import (
    "bytes"
    "context"
    "errors"
    "flag"
    "fmt"
    "io"
    "net/http"
    "path"
    "strings"
    "sync"
    "time"

    "github.com/golang/snappy"

    "github.com/VictoriaMetrics/VictoriaMetrics/lib/cgroup"

    "github.com/VictoriaMetrics/VictoriaMetrics/lib/httputil"
    "github.com/VictoriaMetrics/VictoriaMetrics/lib/logger"
    "github.com/VictoriaMetrics/VictoriaMetrics/lib/netutil"

    "github.com/VictoriaMetrics/VictoriaMetrics/lib/promauth"
    "github.com/VictoriaMetrics/VictoriaMetrics/lib/prompb"
    "github.com/VictoriaMetrics/metrics"
)

var defaultConcurrency = cgroup.AvailableCPUs() * 2
```



```

const (
    defaultMaxBatchSize = 1e4
    defaultMaxQueueSize = 1e5
    defaultFlushInterval = 2 * time.Second
    defaultWriteTimeout = 30 * time.Second
)

var (
    disablePathAppend =
flag.Bool("remoteWrite.disablePathAppend", false,
    "Whether to disable automatic appending of
    '/api/v1/write' path to the configured -remoteWrite.url.")
    sendTimeout =
flag.Duration("remoteWrite.sendTimeout",
    30*time.Second, "Timeout for sending data to the
    configured -remoteWrite.url.")
    retryMinInterval =
flag.Duration("remoteWrite.retryMinInterval",
    time.Second, "The minimum delay between retry
    attempts. Every next retry attempt will double the delay
    to prevent hammering of remote database. See also -
    remoteWrite.retryMaxTime")
    retryMaxTime =
flag.Duration("remoteWrite.retryMaxTime",
    time.Second*30, "The max time spent on retry attempts
    for the failed remote-write request. Change this value if
    it is expected for remoteWrite.url to be unreachable for
    more than -remoteWrite.retryMaxTime. See also -
    remoteWrite.retryMinInterval")
)

// Client is an asynchronous HTTP client for writing
// timeseries via remote write protocol.
type Client struct {
    addr      string
    c          *http.Client
    authCfg    *promauth.Config
    input      chan prompb.TimeSeries
    flushInterval time.Duration
    maxBatchSize int
    maxQueueSize int

```

```

    wg    sync.WaitGroup
    doneCh chan struct{}
}

// Config is config for remote write client.
type Config struct {
    // Addr of remote storage
    Addr    string
    AuthCfg *promauth.Config

    // Concurrency defines number of readers that
    // concurrently read from the queue and flush data
    Concurrency int
    // MaxBatchSize defines max number of timeseries
    // to be flushed at once
    MaxBatchSize int
    // MaxQueueSize defines max length of input queue
    // populated by Push method.
    // Push will be rejected once queue is full.
    MaxQueueSize int
    // FlushInterval defines time interval for flushing
    batches
    FlushInterval time.Duration
    // Transport will be used by the underlying http.Client
    Transport *http.Transport
}

// NewClient returns asynchronous client for
// writing timeseries via remotewrite protocol.
func NewClient(ctx context.Context, cfg Config) (*Client,
error) {
    if cfg.Addr == "" {
        return nil, fmt.Errorf("config.Addr can't be empty")
    }
    if cfg.MaxBatchSize == 0 {
        cfg.MaxBatchSize = defaultMaxBatchSize
    }
    if cfg.MaxQueueSize == 0 {
        cfg.MaxQueueSize = defaultMaxQueueSize
    }
}

```

```

    if cfg.FlushInterval == 0 {
        cfg.FlushInterval = defaultFlushInterval
    }
    if cfg.Transport == nil {
        cfg.Transport = httputil.NewTransport(false,
"vmalert_remotewrite")
    }
    cc := defaultConcurrency
    if cfg.Concurrency > 0 {
        cc = cfg.Concurrency
    }
    c := &Client{
        c: &http.Client{
            Timeout: *sendTimeout,
            Transport: cfg.Transport,
        },
        addr:      strings.TrimSuffix(cfg.Addr, "/"),
        authCfg:   cfg.AuthCfg,
        flushInterval: cfg.FlushInterval,
        maxBatchSize: cfg.MaxBatchSize,
        maxQueueSize: cfg.MaxQueueSize,
        doneCh:     make(chan struct{}),
        input:      make(chan prompb.TimeSeries,
cfg.MaxQueueSize),
    }

    for i := 0; i < cc; i++ {
        c.run(ctx)
    }
    return c, nil
}

// Push adds timeseries into queue for writing into
remote storage.
// Push returns and error if client is stopped or if queue is
full.
func (c *Client) Push(s prompb.TimeSeries) error {
    rwTotal.Inc()
    select {
    case <-c.doneCh:
        rwErrors.Inc()

```

```

        droppedRows.Add(len(s.Samples))
        return fmt.Errorf("client is closed")
    case c.input <- s:
        return nil
    default:
        rwErrors.Inc()
        droppedRows.Add(len(s.Samples))
        return fmt.Errorf("failed to push timeseries - queue
is full (%d entries). "+
        "Queue size is controlled by -
remoteWrite.maxQueueSize flag",
        c.maxQueueSize)
    }
}

// Close stops the client and waits for all goroutines
// to exit.
func (c *Client) Close() error {
    if c.doneCh == nil {
        return fmt.Errorf("client is already closed")
    }
    close(c.input)

    start := time.Now()
    logger.Infof("shutting down remote write client:
flushing remained series")
    close(c.doneCh)
    c.wg.Wait()
    logger.Infof("shutting down remote write client:
finished in %v", time.Since(start))

    return nil
}

func (c *Client) run(ctx context.Context) {
    ticker := time.NewTicker(c.flushInterval)
    wr := &prompb.WriteRequest{}
    shutdown := func() {
        lastCtx, cancel :=
context.WithTimeout(context.Background(),
defaultWriteTimeout)

```

```

    for ts := range c.input {
        wr.Timeseries = append(wr.Timeseries, ts)
        if len(wr.Timeseries) >= c.maxBatchSize {
            c.flush(lastCtx, wr)
        }
    }
    // flush the last batch. flush will re-check and avoid
    flushing empty batch.
    c.flush(lastCtx, wr)

    cancel()
}
c.wg.Add(1)
go func() {
    defer c.wg.Done()
    defer ticker.Stop()
    for {
        select {
        case <-c.doneCh:
            shutdown()
            return
        case <-ctx.Done():
            shutdown()
            return
        case <-ticker.C:
            c.flush(ctx, wr)
        case ts, ok := <-c.input:
            if !ok {
                continue
            }
            wr.Timeseries = append(wr.Timeseries, ts)
            if len(wr.Timeseries) >= c.maxBatchSize {
                c.flush(ctx, wr)
            }
        }
    }
}()
}

var (

```

```

    rwErrors =
metrics.NewCounter(vmalert_remotewrite_errors_tot
al)
    rwTotal =
metrics.NewCounter(vmalert_remotewrite_total)

    sentRows      =
metrics.NewCounter(vmalert_remotewrite_sent_rows_
total)
    sentBytes      =
metrics.NewCounter(vmalert_remotewrite_sent_bytes
_total)
    droppedRows    =
metrics.NewCounter(vmalert_remotewrite_dropped_ro
ws_total)
    sendDuration   =
metrics.NewFloatCounter(vmalert_remotewrite_send_d
uration_seconds_total)
    bufferFlushDuration =
metrics.NewHistogram(vmalert_remotewrite_flush_du
ration_seconds)

    _ =
metrics.NewGauge(vmalert_remotewrite_concurrency
, func() float64 {
    return float64(*concurrency)
})
)

// GetDroppedRows returns value of droppedRows metric
func GetDroppedRows() int { return
int(droppedRows.Get()) }

// flush is a blocking function that marshals WriteRequest
and sends
// it to remote-write endpoint. Flush performs limited
amount of retries
// if request fails.
func (c *Client) flush(ctx context.Context, wr
*prompb.WriteRequest) {

```

```

    if len(wr.Timeseries) < 1 {
        return
    }
    defer wr.Reset()
    defer
bufferFlushDuration.UpdateDuration(time.Now())

    data := wr.MarshalProtobuf(nil)
    b := snappy.Encode(nil, data)

    retryInterval, maxRetryInterval := *retryMinInterval,
*retryMaxTime
    if retryInterval > maxRetryInterval {
        retryInterval = maxRetryInterval
    }
    timeStart := time.Now()
    defer func() {
        sendDuration.Add(time.Since(timeStart).Seconds())
    }()
L:
    for attempts := 0; ; attempts++ {
        err := c.send(ctx, b)
        if err != nil && (errors.Is(err, io.EOF) ||
netutil.IsTrivialNetworkError(err)) {
            // Something in the middle between client and
destination might be closing
            // the connection. So we do a one more attempt
in hope request will succeed.
            err = c.send(ctx, b)
        }
        if err == nil {
            sentRows.Add(len(wr.Timeseries))
            sentBytes.Add(len(b))
            return
        }

        _, isNotRetriable := err.(*nonRetriableError)
        logger.Warnf("attempt %d to send request failed:
%s (retriable: %v)", attempts+1, err, !isNotRetriable)

        if isNotRetriable {

```

```

        // exit fast if error isn't retrieable
        break
    }

    // check if request has been cancelled before
    backoff
    select {
    case <-ctx.Done():
        logger.Errorf("interrupting retry attempt %d:
context cancelled", attempts+1)
        break L
    default:
    }

    timeLeftForRetries := maxRetryInterval -
time.Since(timeStart)
    if timeLeftForRetries < 0 {
        // the max retry time has passed, so we give up
        break
    }

    if retryInterval > timeLeftForRetries {
        retryInterval = timeLeftForRetries
    }
    // sleeping to prevent remote db hammering
    time.Sleep(retryInterval)
    retryInterval *= 2

}

rwErrors.Inc()
rows := 0
for _, ts := range wr.Timeseries {
    rows += len(ts.Samples)
}
droppedRows.Add(rows)
logger.Errorf("attempts to send remote-write request
failed - dropping %d time series",
    len(wr.Timeseries))
}

```



```

func (c *Client) send(ctx context.Context, data []byte)
error {
    r := bytes.NewReader(data)
    req, err := http.NewRequestWithContext(ctx,
http.MethodPost, c.addr, r)
    if err != nil {
        return fmt.Errorf("failed to create new HTTP
request: %w", err)
    }

    // RFC standard compliant headers
    req.Header.Set("Content-Encoding", "snappy")
    req.Header.Set("Content-Type", "application/x-
protobuf")

    // Prometheus compliant headers
    req.Header.Set("X-Prometheus-Remote-Write-Version",
"0.1.0")

    if c.authCfg != nil {
        err = c.authCfg.SetHeaders(req, true)
        if err != nil {
            return &nonRetriableError{
                err: err,
            }
        }
    }
    if !*disablePathAppend {
        req.URL.Path = path.Join(req.URL.Path,
"/api/v1/write")
    }
    resp, err := c.c.Do(req)
    if err != nil {
        return fmt.Errorf("error while sending request to
%s: %w; Data len %d(%d)",
req.URL.Redacted(), err, len(data), r.Size())
    }
    defer func() { _ = resp.Body.Close() }()

    body, _ := io.ReadAll(resp.Body)

```

```

// according to
https://prometheus.io/docs/concepts/remote_write_spec/
// Prometheus remote Write compatible receivers
MUST
switch resp.StatusCode / 100 {
case 2:
    // respond with HTTP 2xx status code when write is
    // successful.
    return nil
case 4:
    if resp.StatusCode != http.StatusTooManyRequests
    {
        // MUST NOT retry write requests on HTTP 4xx
        // responses other than 429
        return &nonRetriableError{
            err: fmt.Errorf("unexpected response code %d
            for %s. Response body %q", resp.StatusCode,
            req.URL.Redacted(), body),
        }
    }
    fallthrough
default:
    return fmt.Errorf("unexpected response code %d for
    %s. Response body %q",
    resp.StatusCode, req.URL.Redacted(), body)
}
}

type nonRetriableError struct {
    err error
}

func (e *nonRetriableError) Error() string {
    return e.err.Error()
}

```



已思考 1m 57s

- 使用者提供了 `client.go` 程式碼，而我們之前討論過 503 錯誤。看起來他們疑慮 URL 路徑有問題，並指出 `path.Join(req.URL.Path, "/api/v1/write")` 可能導致問題。要注意，`path.Join` 使用時如果第二個參數有 leading slash，會丟掉之前的路徑，這可能會造成不預期的結果。
- 當 `path.Join` 用於將 `"/api/v1/write"` 與 URL 路徑組合時，因為傳入的 `"/api/v1/write"` 有 leading slash，所以會忽略先前的路徑，最終路徑會變成 `/api/v1/write`。這樣會導致丟失原本的路徑，並使請求發送到錯誤的端點 `http://host/api/v1/write`。
- 這樣的行為會導致 VictoriaMetrics 的 `vminsert` 中斷，因為它需要 `/insert/<account>/prometheus/api/v1/write`。如果看到日志中包含空格的路徑 `/insert/0/prometheus/api/v1 /write`，這會被 `path.Join` 處理為 `/api/v1/write`，忽略了前面的路徑。這很可能不是設計意圖。
- 看來 VictoriaMetrics 可能希望將 `/api/v1/write` 附加到任何路徑，但使用 `path.Join` 時，因為傳入帶有斜杠的路徑，它會重設之前的路徑，這可能是個錯誤，除非他們預期 `req.URL.Path` 在特定情況下是空的。如果基礎 URL 配置錯誤，最終路徑會丟失一部分。
- 看起來事發的問題與 `path.Join` 的行為有關。當 `req.URL.Path` 包含 `/insert/0/prometheus` 並與 `/api/v1/write` 合併時，`path.Join` 會重設路徑，導致原有的路徑丟失。如果希望規避此問題，建議將 `-remoteWrite.disablePathAppend=true` 並使用完整的 URL。這樣會使得路徑正確，並避免問題。
- 1. 設置 `-remoteWrite.disablePathAppend=true` 並給予完整的 URL 來避免 `path.Join` 出現異常行為。
 2. 確保無尾隨斜線或空白符，修剪 `c.addr`。
 3. 提供建議的 `vmalert` 參數設定，如調整 `sendTimeout` 和 `retryMaxTime`。
 4. 檢查 Istio 配置與 Envoy 行為問題。
 5. 測試最簡單的 `curl` 呼叫。

☑ 完成

太好了，你把 `vmalert/remotewrite/client.go` 貼出來，關鍵行都在這裡，直接對照你的 503 來解：

這段程式對行為的影響（重點 4 條）

1. 會自動補 ``/api/v1/write``

```
var disablePathAppend = flag.Bool("remoteWrite.disablePathAppend", false,
...
...
if !*disablePathAppend {
    req.URL.Path = path.Join(req.URL.Path, "/api/v1/write")
}
```

也就是：除非你加 ``-remoteWrite.disablePathAppend=true``，否則 `vmaalert` 會把你設定的 ``-remoteWrite.url`` 的 `Path` 後面，再接一個 ``/api/v1/write``。

2. Timeout 與重試行為

```
sendTimeout = flag.Duration("remoteWrite.sendTimeout", 30*time.Second, ...)
retryMinInterval=1s, retryMaxTime=30s (倍增回退)
```

送出逾時預設 30s；失敗會以 1s 起跳、倍增回退、最多重試 30s。超過就丟棄該批（並記 ``droppedRows``）。

3. Flush 與批次

``defaultFlushInterval=2s``、``defaultMaxBatchSize=1e4``、``defaultMaxQueueSize=1e5``。也就是每 ~2 秒沖一次、每批最多 1 萬條、佇列滿就丟。

4. 實際發送的 URL 會被記錄

Log 用 ``req.URL.Redacted()`` 印出「實際要打的 URL」。你看到的

```
.../insert/0/prometheus/api/v1 /write
```

中間多了一個空白，高度懷疑是你配置的 URL（或 Helm/環境變數）含了不可見空白或換行，不是這段程式碼自己加的（程式碼固定補的是 ``/api/v1/write``，不會夾空白）。

跟你目前 503 的關係

- 503 訊息是 **Envoy/Istio** 在拿到上游 **header** 前連線就被關（前面我已解釋）。
- 你的路徑中出現 ``api/v1`/write``（``v1`` 和 ``/write`` 之間有空白），很容易讓上游拒絕或路由不到正確 **handler**，再加上 sidecar 連線池/超時不合，就會出現你看到的 **retriable 503**；隨後重試成功，所以資料還是進到 `vmstorage`。

兩個「一槍斃命」修法（先做第 1 個）

方法 A（最穩）：關掉自動補 path，手動給完整路徑

把 `vmalert` 的參數改成完整的寫入端點，並關掉自動補：

```
-remoteWrite.url=http://vminsert.  
<ns>.svc.cluster.local:8480/insert/0/prometheus/api/v1/write  
-remoteWrite.disablePathAppend=true
```

“這樣就完全繞過 `path.Join(..., "/api/v1/write")``，也避免任何因為 `Join` 或不可見字元導致的怪路徑。”

方法 B（可行）：讓 `vmalert` 自己補 ``/api/v1/write``，但把底座寫乾淨

若你想保留預設補路徑機制，底座 **URL** 只寫到 ``/insert/0/prometheus``，且保證沒有空白/換行：

```
-remoteWrite.url=http://vminsert.  
<ns>.svc.cluster.local:8480/insert/0/prometheus  
# 保持預設：-remoteWrite.disablePathAppend=false
```

“注意：請不要自己在 URL 後面再寫一次 ``/api/v1/write``，讓它自動補就好；而且要確定沒有多餘空白。”

一次到位的建議參數（含穩定化）

```
# 路徑建議（擇一）：  
-remoteWrite.url=http://vminsert.  
<ns>.svc.cluster.local:8480/insert/0/prometheus/api/v1/write  
-remoteWrite.disablePathAppend=true  
  
# or （若不想關 auto-append）  
# -remoteWrite.url=http://vminsert.  
<ns>.svc.cluster.local:8480/insert/0/prometheus  
# -remoteWrite.disablePathAppend=false # 預設  
  
# 時間參數(與 Istio 對齊，降低誤判 503)  
-remoteWrite.sendTimeout=60s  
-remoteWrite.retryMinInterval=1s  
-remoteWrite.retryMaxTime=60s  
  
# 佇列/批次（看量能微調）  
-remoteWrite.maxBatchSize=10000  
-remoteWrite.maxQueueSize=100000  
  
# 降低並行請求（Envoy/後端負載高時很有用）  
-remoteWrite.concurrency=1
```

立刻檢查 3 件事（很快）

1. 是否有不可見字元

在 vmalert Pod 內：

```
echo -n "$REMOTEWRITE_URL" | hexdump -C  
# 或把實際參數 echo 出來檢查，注意結尾是否出現 0x20（空白）或 0x0a（換行）
```

2. 用 **ClusterIP** 直連健康檢查（先排除 Istio）

```
curl -sS -v http://vminsert.<ns>.svc.cluster.local:8480/health
```

3. 抓 **vmalet** 生效的 **URL**

vmalet 的 warn 裡會印 ``... for http://<host>/...``，修正參數後再觀察是否成為**沒有空白**的正確路徑。

與前一則建議的 **Istio** 設定如何配合

- 如果你**必須**走 VirtualService/Gateway，請把 VS/DR 的 timeout、retry、連線池、``h2UpgradePolicy: DO_NOT_UPGRADE`` 等照我上一則給的範例放寬。
- 但在你找出 **URL 路徑的空白/換行** 之前，**優先**用 ClusterIP 直連 + 方法 A 先止血。

小結

- 這份程式碼告訴我們：**預設會自動補 ``/api/v1/write``**，而你 log 裡的多餘空白極可能來自 **配置文字**，不是程式碼。
- 先用 ``-remoteWrite.disablePathAppend=true`` + **完整 URL** 把路徑鎖死，並把 ``sendTimeout/retryMaxTime`` 稍微放寬；503 會明顯下降。
- 若仍有零星 503，再回到 Istio 的 VS/DR 參數與連線池做微調即可。

如果你願意，把你現在的 vmalet **實際參數（或 Helm values）** 那一段貼上來，我幫你把 ``url/disablePathAppend`` 與 timeout 等參數整理成正確版本。



並把 `sendTimeout/retryMaxTime` 稍微放寬；503 會明顯下降，這怎麼做



已思考 38s

- 我看到用戶需要具體的步驟來調整vmalet遠程寫入的參數，基於K8s環境。這包括如何設置Helm值、Deployment/StatefulSet中的參數、環境變量，並提供相應的YAML修補、命令行旗標。需要考慮調整Istio的代理設置，使其與超時協調一致。
- 這裡的目標是調整flush間隔與sendTimeout的互動：flush每2秒發送批次；sendTimeout越大，等待時間越長。應該監控的指標、如何通過日誌與指標驗證也很重要。vMinert應該迅速響應，VictoriaMetrics的遠程寫入速度很快。
- 提供修改命令示例，可以使用 ``kubectl set args`` 或 ``kubectl apply -f`` 更安全的建議來設置 vmalet 的相關參數。要注意使用 ``kubectl set args`` 會替換整個參數列表，這有風險。也可以通過 Helm 值來設置，例如 ``vmalet.extraArgs``。
- 總的flush週期包括最壞情況的延遲，通常是sendTimeout加上最大時間的重試回退。通過設置sendTimeout和retryMaxTime來避免積壓，建議sendTimeout為60秒，retryMaxTime在60-90秒之間。對於virtualService，可將perTryTimeout設置為10秒，並略微調整以避免超時衝突。

🕒 完成

可以，直接改 vmalet 的 **旗標參數** 就好。這兩個對應到你貼的程式碼裡的全域 flags：

- ``-remoteWrite.sendTimeout``（預設 30s）
- ``-remoteWrite.retryMaxTime``（預設 30s）

下面給你三種常見做法，擇一使用即可。



1) 直接改 Deployment/StatefulSet (最通用)

把 vmaalert 容器的 `args:` 補上兩個旗標 (建議一起把 URL 鎖死，避免自動補路徑有空白)：

```
# vmaalert Deployment/StatefulSet 片段
spec:
  template:
    spec:
      containers:
        - name: vmaalert
          args:
            - -remoteWrite.url=http://vminsert.
<ns>.svc.cluster.local:8480/insert/0/prometheus/api/v1/write
            - -remoteWrite.disablePathAppend=true
            - -remoteWrite.sendTimeout=60s
            - -remoteWrite.retryMaxTime=60s
            - -remoteWrite.retryMinInterval=1s
# 其餘既有參數照舊保留
```

“套用：`kubectl -n <ns> apply -f vmaalert.yaml`”

2) 用 `kubectl set args` (快速改，但要小心覆蓋)

這個指令會「取代整個 args」，所以務必把原本的所有參數一起重寫上去：

```
kubectl -n <ns> set args deploy/vmaalert --containers=vmaalert -- \
  -remoteWrite.url=http://vminsert.
<ns>.svc.cluster.local:8480/insert/0/prometheus/api/v1/write \
  -remoteWrite.disablePathAppend=true \
  -remoteWrite.sendTimeout=60s \
```

```
-remoteWrite.retryMaxTime=60s \  
-remoteWrite.retryMinInterval=1s \  
<把你原本的其餘 vmalert 旗標也一併補上>
```

3) Helm chart (如果你是用 Helm 部署)

不同 chart 寫法略有差異，通常是 `extraArgs`，會支援 **map** 或 **list** 兩種風格之一，你用你 chart 支援的那種：

map 風格：

```
vmalert:  
  extraArgs:  
    remoteWrite.url: "http://vminsert.  
<ns>.svc.cluster.local:8480/insert/0/prometheus/api/v1/write"  
    remoteWrite.disablePathAppend: "true"  
    remoteWrite.sendTimeout: "60s"  
    remoteWrite.retryMaxTime: "60s"  
    remoteWrite.retryMinInterval: "1s"
```

list 風格：

```
vmalert:  
  extraArgs:  
    - remoteWrite.url=http://vminsert.  
<ns>.svc.cluster.local:8480/insert/0/prometheus/api/v1/write  
    - remoteWrite.disablePathAppend=true  
    - remoteWrite.sendTimeout=60s  
    - remoteWrite.retryMaxTime=60s  
    - remoteWrite.retryMinInterval=1s
```

“套用：`helm upgrade --install <release> <chart> -n <ns> -f values.yaml`”

為什麼用 60s？

- 你的 503 多半是連線還沒拿到 **header** 就被中間層關掉造成的抖動；把 `sendTimeout` 拉到 **60s**，加上 `retryMaxTime` **60s**，重試窗變長，赫然間偶發 **503** 就被吸收（常見在滾更、瞬時負載、proxy 轉換時發生）。
- `retryMinInterval` 設 **1s**（預設也是 1s），維持倍增回退即可。

若你必須經過 Istio/Envoy（VirtualService）

把 VS 的 timeout 調高，避免 proxy 先關連線、vmalet 還在等：

```
# VirtualService 片段 (例)
retries:
  attempts: 2
  perTryTimeout: 55s      # <= sendTimeout(60s)
  retryOn: 5xx,connect-failure,reset,refused-stream,gateway-error,stream-reset
  timeout: 75s            # > sendTimeout(60s), 也涵蓋 2 次嘗試
```

“DestinationRule 建議 `h2UpgradePolicy: DO_NOT_UPGRADE`、放寬 `connectionPool` 與 `idleTimeout`（如我上一則回覆的範本）。”

套用後怎麼確認？

1. 看 vmalet log：`attempt X to send request failed` 的頻率應下降；URL 路徑應該沒有空白。

2. 查指令列參數是否生效：

```
kubectl -n <ns> exec -it deploy/vmalert -- sh -c 'ps ax | grep vmalert | tr "\0" " "'
```

3. 觀察 metrics：

- `vmalert_remotewrite_errors_total` 成長變慢/停滯
- `vmalert_remotewrite_dropped_rows_total` 不再上升

需要的話你把現在 vmalert 的 Deployment/Helm values 那段貼上來，我直接幫你改成可貼用的版本（保留你原本其它旗標）。



使用的是 K8s

vminsert有掛VirtualService、Service，vmalert的-remoteWrite.url是vminsert的VirtualService

這是出現在vmalert寫入alert到vminsert時出現的log，503，可是vminsert 有把alert寫進vmstorage

vmalert log出現這個，我找不到處理方法：

2025-08-01T16:56:03.749+0800

warn

VictoriaMetrics/app/ vmaalert/remotewrite/client.go:

256

attempt 1 to send

request failed: unexpected response code 503 for http://
vm-vminsert.ifond.edg.f18.tsmc

.com/insert/0/prometheus/api/v1 /write. Response body
"upstream connect error or disconnect/reset before
headers. reset reason: connection termination"
(retriable: true)2025-08-01

這是VictoriaMetrics/app/ vmaalert/remotewrite/client.go:

package remotewrite

import (

 "bytes"

 "context"

 "errors"

 "flag"

 "fmt"

 "io"

 "net/http"

 "path"

```
"strings"

"sync"

"time"


"github.com/golang/snappy"


"github.com/VictoriaMetrics/VictoriaMetrics/lib/cgroup"
"github.com/VictoriaMetrics/VictoriaMetrics/lib/logger"
"github.com/VictoriaMetrics/VictoriaMetrics/lib/netutil"


"github.com/VictoriaMetrics/VictoriaMetrics/lib/promauth
"

"github.com/VictoriaMetrics/VictoriaMetrics/lib/prompbmarshal"

"github.com/VictoriaMetrics/metrics"
)

var defaultConcurrency = cgroup.AvailableCPUs() * 2

const (

    defaultMaxBatchSize = 1e4

    defaultMaxQueueSize = 1e5
```

```

        defaultFlushInterval = 2 * time.Second

        defaultWriteTimeout = 30 * time.Second

    )

    var (

        disablePathAppend =
            flag.Bool("remoteWrite.disablePathAppend", false,
                "Whether to disable automatic appending of
                '/api/v1/write' path to the configured -remoteWrite.url.")

        sendTimeout =
            flag.Duration("remoteWrite.sendTimeout",
                30*time.Second, "Timeout for sending data to the
                configured -remoteWrite.url.")

        retryMinInterval =
            flag.Duration("remoteWrite.retryMinInterval",
                time.Second, "The minimum delay between retry
                attempts. Every next retry attempt will double the delay
                to prevent hammering of remote database. See also -
                remoteWrite.retryMaxTime")

        retryMaxTime =
            flag.Duration("remoteWrite.retryMaxTime",
                time.Second*30, "The max time spent on retry attempts
                for the failed remote-write request. Change this value if
                it is expected for remoteWrite.url to be unreachable for
                more than -remoteWrite.retryMaxTime. See also -
                remoteWrite.retryMinInterval")

    )

    // Client is an asynchronous HTTP client for writing

```

```
// timeseries via remote write protocol.

type Client struct {

    addr      string

    c          *http.Client

    authCfg    *promauth.Config

    input      chan prompbmarshal.TimeSeries

    flushInterval time.Duration

    maxBatchSize int

    maxQueueSize int

    wg         sync.WaitGroup

    doneCh chan struct{ }

}
```

```
// Config is config for remote write client.
```

```
type Config struct {

    // Addr of remote storage

    Addr string

    AuthCfg *promauth.Config
```



```

// Concurrency defines number of readers that
// concurrently read from the queue and flush data
Concurrency int

// MaxBatchSize defines max number of timeseries
// to be flushed at once
MaxBatchSize int

// MaxQueueSize defines max length of input queue
// populated by Push method.
// Push will be rejected once queue is full.
MaxQueueSize int

// FlushInterval defines time interval for flushing
batches
FlushInterval time.Duration

// Transport will be used by the underlying http.Client
Transport *http.Transport
}

// NewClient returns asynchronous client for
// writing timeseries via remotewrite protocol.
func NewClient(ctx context.Context, cfg Config) (*Client,
error) {

    if cfg.Addr == "" {

```

```
        return nil, fmt.Errorf("config.Addr can't be empty")
    }

    if cfg.MaxBatchSize == 0 {
        cfg.MaxBatchSize = defaultMaxBatchSize
    }

    if cfg.MaxQueueSize == 0 {
        cfg.MaxQueueSize = defaultMaxQueueSize
    }

    if cfg.FlushInterval == 0 {
        cfg.FlushInterval = defaultFlushInterval
    }

    if cfg.Transport == nil {
        cfg.Transport = http.DefaultTransport.
(*http.Transport).Clone()
    }

    cc := defaultConcurrency

    if cfg.Concurrency > 0 {
        cc = cfg.Concurrency
    }

    c := &Client{

        c: &http.Client{
```

```

        Timeout: *sendTimeout,

        Transport: cfg.Transport,

    },

    addr:      strings.TrimSuffix(cfg.Addr, "/"),

    authCfg:   cfg.AuthCfg,

    flushInterval: cfg.FlushInterval,

    maxBatchSize: cfg.MaxBatchSize,

    maxQueueSize: cfg.MaxQueueSize,

    doneCh:     make(chan struct{}),

    input:      make(chan prompbmarshal.TimeSeries,
cfg.MaxQueueSize),

    }

    for i := 0; i < cc; i++ {

        c.run(ctx)

    }

    return c, nil

}

// Push adds timeseries into queue for writing into
remote storage.

```

// Push returns and error if client is stopped or if queue is full.

```
func (c *Client) Push(s prompbmarshal.TimeSeries) error
{
    rwTotal.Inc()

    select {

    case <-c.doneCh:

        rwErrors.Inc()

        droppedRows.Add(len(s.Samples))

        return fmt.Errorf("client is closed")

    case c.input <- s:

        return nil

    default:

        rwErrors.Inc()

        droppedRows.Add(len(s.Samples))

        return fmt.Errorf("failed to push timeseries - queue
is full (%d entries). "+

            "Queue size is controlled by -
remoteWrite.maxQueueSize flag",

            c.maxQueueSize)

    }
}
```

```
// Close stops the client and waits for all goroutines
```

```
// to exit.
```

```
func (c *Client) Close() error {
```

```
    if c.doneCh == nil {
```

```
        return fmt.Errorf("client is already closed")
```

```
    }
```

```
    close(c.input)
```

```
    close(c.doneCh)
```

```
    c.wg.Wait()
```

```
    return nil
```

```
}
```

```
func (c *Client) run(ctx context.Context) {
```

```
    ticker := time.NewTicker(c.flushInterval)
```

```
    wr := &prompbmarshal.WriteRequest{}
```

```
    shutdown := func() {
```

```
        lastCtx, cancel :=  
context.WithTimeout(context.Background(),  
defaultWriteTimeout)
```

```
        logger.Infof("shutting down remote write client and  
flushing remained series")
```

```

shutdownFlushCnt := 0

for ts := range c.input {

    wr.Timeseries = append(wr.Timeseries, ts)

    if len(wr.Timeseries) >= c.maxBatchSize {

        shutdownFlushCnt += len(wr.Timeseries)

        c.flush(lastCtx, wr)

    }

}

// flush the last batch. flush will re-check and avoid
flushing empty batch.

shutdownFlushCnt += len(wr.Timeseries)

c.flush(lastCtx, wr)

logger.Infof("shutting down remote write client
flushed %d series", shutdownFlushCnt)

cancel()

}

c.wg.Add(1)

go func() {

    defer c.wg.Done()

    defer ticker.Stop()

```

```
for {  
    select {  
  
        case <-c.doneCh:  
            shutdown()  
  
            return  
  
        case <-ctx.Done():  
            shutdown()  
  
            return  
  
        case <-ticker.C:  
            c.flush(ctx, wr)  
  
        case ts, ok := <-c.input:  
            if !ok {  
                continue  
            }  
  
            wr.Timeseries = append(wr.Timeseries, ts)  
  
            if len(wr.Timeseries) >= c.maxBatchSize {  
                c.flush(ctx, wr)  
            }  
        }  
    }  
}  
  
}()
```

```

}

var (
    rwErrors =
metrics.NewCounter(vmaalert_remotewrite_errors_tot
al)

    rwTotal =
metrics.NewCounter(vmaalert_remotewrite_total)

    sentRows      =
metrics.NewCounter(vmaalert_remotewrite_sent_rows_
total)

    sentBytes      =
metrics.NewCounter(vmaalert_remotewrite_sent_bytes
_total)

    droppedRows    =
metrics.NewCounter(vmaalert_remotewrite_dropped_ro
ws_total)

    sendDuration   =
metrics.NewFloatCounter(vmaalert_remotewrite_send_d
uration_seconds_total)

    bufferFlushDuration =
metrics.NewHistogram(vmaalert_remotewrite_flush_du
ration_seconds)

    _ =
metrics.NewGauge(vmaalert_remotewrite_concurrency

```



```

, func() float64 {

    return float64(*concurrency)

})

)

// GetDroppedRows returns value of droppedRows metric

func GetDroppedRows() int64 { return
int64(droppedRows.Get()) }

// flush is a blocking function that marshals WriteRequest
and sends

// it to remote-write endpoint. Flush performs limited
amount of retries

// if request fails.

func (c *Client) flush(ctx context.Context, wr
*prompbmarshal.WriteRequest) {

    if len(wr.Timeseries) < 1 {

        return

    }

    defer wr.Reset()

    defer
bufferFlushDuration.UpdateDuration(time.Now())

```

```

data := wr.MarshalProtobuf(nil)

b := snappy.Encode(nil, data)

retryInterval, maxRetryInterval := *retryMinInterval,
*retryMaxTime

if retryInterval > maxRetryInterval {
    retryInterval = maxRetryInterval
}

timeStart := time.Now()

defer func() {
    sendDuration.Add(time.Since(timeStart).Seconds())
}()

L:

for attempts := 0; ; attempts++ {

    err := c.send(ctx, b)

    if err != nil && (errors.Is(err, io.EOF) ||
netutil.IsTrivialNetworkError(err)) {

        // Something in the middle between client and
destination might be closing

        // the connection. So we do a one more attempt
in hope request will succeed.

        err = c.send(ctx, b)

    }

```

```

    if err == nil {

        sentRows.Add(len(wr.Timeseries))

        sentBytes.Add(len(b))

        return

    }

    _, isNotRetriable := err.(*nonRetriableError)

    logger.Warnf("attempt %d to send request failed:
    %s (retriable: %v)", attempts+1, err, !isNotRetriable)

    if isNotRetriable {

        // exit fast if error isn't retriable

        break

    }

    // check if request has been cancelled before
    backoff

    select {

    case <-ctx.Done():

        logger.Errorf("interrupting retry attempt %d:
        context cancelled", attempts+1)

        break L

```

```

        default:

    }

    timeLeftForRetries := maxRetryInterval -
time.Since(timeStart)

    if timeLeftForRetries < 0 {

        // the max retry time has passed, so we give up

        break

    }

    if retryInterval > timeLeftForRetries {

        retryInterval = timeLeftForRetries

    }

    // sleeping to prevent remote db hammering

    time.Sleep(retryInterval)

    retryInterval *= 2

}

rwErrors.Inc()

rows := 0

```

```

    for _, ts := range wr.Timeseries {

        rows += len(ts.Samples)

    }

    droppedRows.Add(rows)

    logger.Errorf("attempts to send remote-write request
failed - dropping %d time series",

        len(wr.Timeseries))

}

func (c *Client) send(ctx context.Context, data []byte)
error {

    r := bytes.NewReader(data)

    req, err := http.NewRequestWithContext(ctx,
http.MethodPost, c.addr, r)

    if err != nil {

        return fmt.Errorf("failed to create new HTTP
request: %w", err)

    }

    // RFC standard compliant headers

    req.Header.Set("Content-Encoding", "snappy")

    req.Header.Set("Content-Type", "application/x-
protobuf")

```

```

// Prometheus compliant headers

req.Header.Set("X-Prometheus-Remote-Write-Version",
"0.1.0")

if c.authCfg != nil {

    err = c.authCfg.SetHeaders(req, true)

    if err != nil {

        return &nonRetriableError{

            err: err,

        }

    }

}

if !*disablePathAppend {

    req.URL.Path = path.Join(req.URL.Path,
"/api/v1/write")

}

resp, err := c.c.Do(req)

if err != nil {

    return fmt.Errorf("error while sending request to
%s: %w; Data len %d(%d)",

req.URL.Redacted(), err, len(data), r.Size())

```

```

    }

    defer func() { _ = resp.Body.Close() }()

    body, _ := io.ReadAll(resp.Body)

    // according to
    https://prometheus.io/docs/concepts/remote_write_spec/

    // Prometheus remote Write compatible receivers
    MUST

    switch resp.StatusCode / 100 {

    case 2:

        // respond with HTTP 2xx status code when write is
        successful.

        return nil

    case 4:

        if resp.StatusCode != http.StatusTooManyRequests
        {

            // MUST NOT retry write requests on HTTP 4xx
            responses other than 429

            return &nonRetriableError{

                err: fmt.Errorf("unexpected response code %d
                for %s. Response body %q", resp.StatusCode,
                req.URL.Redacted(), body),

            }
        }
    }

```

```

    }

    fallthrough

default:

    return fmt.Errorf("unexpected response code %d for
%s. Response body %q",

        resp.StatusCode, req.URL.Redacted(), body)

    }
}

type nonRetriableError struct {

    err error

}

func (e *nonRetriableError) Error() string {

    return e.err.Error()

}

```



請問你是透過 YAML（如 Deployment）、Helm，還是其它方式部署 vmaalert？

我會根據你使用的方式，給出具體的修改方式來放寬 `-remoteWrite.sendTimeout`` 和 `-remoteWrite.retryMaxTime``。

如果你能貼上你目前的 `vmaalert` 啟動參數（如 `args` 或 `command` 部分），會更快幫你寫出正確的設定。



Deployment、Helm



了解。我要幫你撰寫 `vmaalert` 使用 Helm 部署時，如何在 `values.yaml` 中正確設定 ``-remoteWrite.sendTimeout`` 和 ``-remoteWrite.retryMaxTime``，並確保 Deployment 中 `args` 傳遞正確。

我會查找 VictoriaMetrics `vmaalert` Helm chart 的參數名稱與格式，並提供一個完整可參考的設定範例（含 `disablePathAppend` 建議），稍後回覆你。

Research completed in 10m·9 個資料來源·56 次搜尋

🔍 `vmaalert` Helm 部署中設定 `sendTimeout` 與 `retryMaxTime`



排查 VMAAlert 写入 VMinsert 出现 503 错误的问题

背景

您在 Kubernetes 集群中部署了 VictoriaMetrics 的集群版组件，包括 **`vmaalert`**（用于执行告警规则）和 **`vminsert`**（用于接收时序数据写入）。其中，``vminsert`` 暴露了一个 Istio `VirtualService`（以及对应的 `Service`），而 ``vmaalert`` 的启动参数 ``-remoteWrite.url`` 配置为指向该 ``vminsert`` 的 `VirtualService` 域名地址。

在此架构下，`vmalert` 会通过 **HTTP Remote Write** 协议将告警状态数据写入 `vminsert`，再由 `vminsert` 存入底层的 `vmstorage`。您遇到的问题是：`vmalert` 日志经常出现 **HTTP 503 错误**，内容为 *“unexpected response code 503 ... upstream connect error or disconnect/reset before headers. reset reason: connection termination”*，并提示请求可重试 (retriable: true)。尽管出现 503 错误，您观察到 `vminsert` 实际上**仍然成功将告警数据写入 `vmstorage`**。换言之，503 错误似乎是暂时性的，`vmalert` 最终重试后数据写入成功，但您希望了解其原因并找到解决方法。

错误含义

日志中的**“upstream connect error or disconnect/reset before headers. reset reason: connection termination”是由网络代理层返回的错误消息，表示到上游服务的连接在发送 HTTP 头部之前就被中断或复位**。在 Istio 服务网格环境中，Envoy 代理使用特定标志记录错误类型：例如`response_flags: "UC"`表示 **Upstream Connection Termination**（上游连接终止） [web.archive.org](https://web.archive.org/web/20200602120000/https://istio.io/docs/reference/config/analysiserrortypes/UpstreamConnectionTermination.html)。简而言之，`vmalert` 发往 `vminsert` 的请求没有正常建立完 HTTP 通信就被中断，这通常意味着网络层或上游服务在请求开始时就关闭了连接。

HTTP 503 Service Unavailable 通常表明请求未能到达目标服务或被代理拦截。结合错误信息来看，这很可能是**Istio VirtualService/Envoy 在转发请求时发生了连接中断**。常见原因包括：目标服务不可达、无可后端实例、连接被拒绝，或者 TLS 握手失败等。

可能原因分析

根据您提供的环境信息和日志，这个503错误很可能与**Istio服务网格配置**有关，具体可能原因包括：

- **主机名与Istio安全策略不匹配**：您使用了 VirtualService 定义的域名（例如 `vm-vminsert.ifond.edg.f18.tsmc.com`）让 `vmalert` 访问 `vminsert`。如果该域名并非 Kubernetes 服务的原始域名，Istio 在启用 mTLS 时可能会遇到证书校验问题。例如，有用户在 Istio 中使用自定义主机名访问集群内服务时，因主机名与服务实际 DNS 名称不一致，导致 mTLS 证书验证失败，从而出现与您类似的

“connection termination” 503 错误 web.archive.org。Istio 默认证书的主旨包含服务帐户和 `.svc.cluster.local` 域名。如果请求主机名不在证书允许的范围内，握手会被拒绝 web.archive.org。在您案例中，`vm-vminsert.ifond.edg.f18.tsmc.com` 可能不是服务的默认域名（通常 Kubernetes 服务域名类似 `vminsert.<命名空间>.svc.cluster.local`），从而引发 mTLS 验证问题。

- **VirtualService路由或服务配置问题:** 如果 VirtualService 配置不当，也可能导致间歇性503。例如 VirtualService 的主机、路由端口、或者 DestinationRule 不匹配，会使 Envoy 找不到正确的上游，从而返回503。另外，如果 VirtualService 依赖 Istio IngressGateway 进行路由，而 `vmalert` 仍通过该网关访问（哪怕在集群内部），可能引入不必要的网络跳转和超时。
- **连接空闲或断开:** 环境中的网络负载均衡或代理可能会断开长时间空闲的连接。当 `vmalert` 定期推送数据时，如果某段时间没有数据发送，先前的TCP连接可能被闲置关闭。下一次发送时，旧连接已失效，导致第一次尝试返回类似 “*upstream connect error*” 的错误。VictoriaMetrics 的团队实际上注意到了**短连接或负载均衡不支持 keep-alive**场景会导致此类日志，因而在新版本中实现了自动重试以减少日志噪音 docs.victoriametrics.com。您的情况很符合这种模式：第一次发送遇到连接断开告警，vmalert 随即重试建立新连接并成功。
- **服务实例重启或瞬时不可用:** 如果 `vminsert` 实例当时正好重启、滚动更新或临时不可达，也会导致 `vmalert` 第一次连接失败返回503，但很快实例恢复，重试成功。虽然这种情况不应频繁发生，但值得核查 `vminsert` Pod 的重启记录或资源使用情况。

综合来看，**Istio 主机名/mTLS 配置不匹配** 是一个重要的可能原因，因为有直接的类似案例佐证。如果 `vmalert` 将 `vminsert` 的 VirtualService域名视作外部主机，可能导致流量未用mTLS加密或者路由不到正确的内部服务，引发连接终止。web.archive.org 提供的Istio讨论中指出，当请求使用与服务实际DNS不符的自定义域名且开启mTLS时，会出现证书验证失败的503错误；改用正确的内部DNS名称请求则一切正常 web.archive.org。这与您的症状相符。

排查与解决方法

针对上述可能原因，建议按以下步骤进行排查并尝试解决：

1. 使用内部服务地址直连（避免通过VirtualService域名）

最快速的验证方法是让 `vmalert` 绕过 `VirtualService`，直接使用 `Kubernetes` 服务的内部地址进行远程写入。如果 `vmalert` 跑在同一集群内，您可以将 `-remoteWrite.url` 改为 `vminsert` 服务的 `ClusterIP` 地址或内部DNS。如：

```
-remoteWrite.url=http://vminsert.<命名空间>.svc.cluster.local:<端口>/insert/0/prometheus
```

（注意不要包含 `/api/v1/write`，`vmalert` 默认会自动附加该路径，除非您显式使用了 `-remoteWrite.disablePathAppend`）。

通过这种方式，流量将在集群内部直接由 `Envoy Sidecar` 解析到 `vminsert` 服务，无需经过 `Istio Ingress Gateway` 或外部DNS。社区经验表明，使用 `Kubernetes` 原生服务地址可以避免因自定义域名导致的TLS主机名不匹配问题 web.archive.org。如果切换后503错误不再出现，基本可以确认是此前使用 `VirtualService` 域名导致的 `Istio` 路由或TLS问题。届时您可以考虑继续使用内部地址作为长期方案，因为在同集群内这样做没有问题，也更简单可靠。

2. 调整Istio配置匹配主机名（如需保留VirtualService）

如果出于架构需要（例如多集群统一配置、或必须通过Ingress流量）而必须使用 `VirtualService` 提供的自定义域名，则需要调整 `Istio` 配置以消除主机名不匹配和连接中断的问题：

- **确保 Service 和 VirtualService 对应**：可以创建一个与自定义域名匹配的 `Kubernetes Service` 或 `ServiceEntry`，使 `Istio` 将该域名识别为内部服务。比如，在上述 `Istio` 论坛案例中，用户通过创建一个类型为 `ExternalName` 的 `Service` 来指向真实服务，并结合 `VirtualService` 完成流量改写，最终解决了问题 web.archive.org。这样 `Envoy` 就不会把 `vm-vminsert.ifond.edg.f18.tsmc.com` 当成未知外部主机。
- **VirtualService Host 和 Host 头重写**：在 `VirtualService` 配置中，使用 `hosts:` 字段包含您的域名，并在路由规则中通过 `rewrite/authority` 将 `Host` 头修改为内部服务的实际主机名。例如，`Istio` 建议的配置是将请求的 `Host` 改写为 `vminsert.<ns>.svc.cluster.local`，然后目标路由也是这个内部 `host`

web.archive.org web.archive.org。这样一来，请求在 Envoy 看来就与直接请求内部服务无异，可通过mTLS校验。确保 DestinationRule 中对该主机开启了 ISTIO_MUTUAL 模式，以使用双向TLS。

- **检查TLS证书和模式**：如果您的Mesh全局开启了mTLS严格模式，务必保证上述配置后客户端Envoy会使用正确的SNI/Host与服务端握手。Istio mTLS验证默认基于SPIFFE身份，但也需要匹配服务发现的主机。您可以在 DestinationRule 中为该流量指定 `trafficPolicy.tls.mode: ISTIO_MUTUAL`（通常默认即可）或相应的 `subjectAltNames` 以匹配服务账户。如果一时难以调整，也可以临时将目的地流量策略改为明文（DISABLE）测试是否握手问题消失，但长期应修正配置而不是禁用安全。

调整配置后，请重新部署并留意 `vmalert` 日志是否还有503错误。如果仍有错误，可以进一步查看Istio代理日志（使用 `istioctl proxy-config log` 或直接进入sidecar容器）以获取详细信息。特别关注 Envoy 日志中的 **response_flags**（如 UC、UF 等）和 **certificate** 相关的调试信息。根据Istio文档，`uc` 表示上游连接被意外终止

droid.io；如果有TLS握手失败通常会有 `'TLS error'` 的日志或以 `NR` (No Route) 标记的503。确认是哪种情况有助于定位问题根因。

3. 检查服务可用性和连接设置

虽然从您的描述看，`vminsert` 实际写入了数据，说明服务本身在重试时是可用的，但也值得检查：

- **`vminsert` 副本数和健康状态**：如果有多个 `vminsert` 实例，503可能是在某一个实例不可用时Envoy选择了它。检查VirtualService/DestinationRule的负载均衡策略以及各Pod的健康状况。如果某Pod经常在健康探针失败后重启，可能导致间歇性503。在这种情况下，应定位 `vminsert` 实例稳定性问题或调整探针/资源。
- **Istio连接池与超时**：Istio有连接池和断开空闲连接的默认行为。如果您的环境空闲连接断开过快，可以考虑在 DestinationRule 中增加 `connectionPool` 设置（例如提高 `idleTimeout`）或者启用重试策略，尽管 vmalert 自身已实现重试。确保 VirtualService/DestinationRule 没有关闭 HTTP Keep-Alive 或异常的超时配置。如果使用Ingress Gateway，也检查 Gateway 的超时是否足够长。

- **网络策略或Pod安全**：确认没有NetworkPolicy或Istio的AuthorizationPolicy阻断`vmalert`到`vminsert`的访问。在Mesh内部通常不需要额外配置，但如果定义了策略，只允许特定主机/端口通信，也可能导致偶发拒绝。

通过上述检查，排除掉基础网络因素后，大概率问题还是围绕Istio域名路由及mTLS。因此重点还是使请求的**Host与服务配置对上**。简单来说，**推荐方案**是直接使用服务的cluster-local 地址，这在实践中被证明可以避免此错误 web.archive.org ；如需用自定义域名，就按照Istio官方建议调整配置来满足mTLS校验 web.archive.org web.archive.org 。

vmalert 的重试机制说明

值得一提的是，`vmalert` 对这种503错误已经有内置的重试逻辑和容错：

- **自动重试**：从您提供的代码可见，`vmalert` 将 5xx 响应视为可重试错误，会进行指数退避的多次尝试。在**v1.97.0**后的版本中，vmalert默认会对失败的Remote Write请求重试最多约30秒（尝试间隔逐渐翻倍），以避免瞬时故障造成数据丢失 docs.victoriametrics.com 。实际上，`vmalert` 项目更新日志中特别提到增加了“**自动重试 Remote Write请求**”的功能，以减少在网络连接短暂中断或负载均衡不支持长连接情况下产生的大量错误日志 docs.victoriametrics.com 。从您的日志看，正是第一次尝试返回503，然后vmalert发出警告并在后续尝试中成功，将数据写入`vminsert`。所以，**数据并未丢失**，只是第一次失败产生了警告日志。
- **可配置参数**：您可以调整`-remoteWrite.sendTimeout`，`-remoteWrite.retryMinInterval`，`-remoteWrite.retryMaxTime`等启动参数（如果有需要）来改变重试行为。例如增大最大重试时长，或者缩短重试间隔。不过一般默认值已足够，大部分情况下更重要的是解决根本网络问题，而非依赖无限重试。
- **队列和批次**：vmalert 对要发送的数据进行缓冲批处理（`maxBatchSize` 默认10000条时间序列）并异步发送。如果下游不通，数据会暂存在内存队列里（最大队列长度默认100k）【VictoriaMetrics vmalert 源码】。但如果长期发送不出去，队列满了也会丢数据。因此，虽然短暂503可重试，无需担心，还是应**尽快解决导致503的根本原因**，以免更严重的网络故障时造成告警数据丢失。

总结

结论： VMAalert 写入 VMinsert 出现的 503 “upstream connect error/connection termination”错误，多半源自 **Istio 环境下的流量路由或TLS配置问题**。

VirtualService提供的自定义域名可能与服务实际DNS名称不一致，触发了Istio mTLS证书验证失败或路由异常 web.archive.org 。建议优先尝试让 vmaalert 使用内部服务地址直连 vminsert web.archive.org ；如果必须用 VirtualService 域名，则需要按Istio最佳实践调整配置（ServiceEntry/VirtualService/DestinationRule）来保证主机名和TLS验证匹配 web.archive.org 。另外，检查Istio代理日志和 vminsert 实例状态，确保无其它网络故障因素。完成上述调整后，您的 vmaalert 日志中应不再出现503错误告警，告警时间序列数据也能更加稳定地写入。由于 vmaalert 自身具备重试机制 docs.victoriametrics.com ，短暂问题不会导致数据丢失，但尽早修复配置可避免日志噪音和潜在隐患。

最后，引用Istio社区对类似问题的描述：“当服务网格启用mTLS时，请求使用的主机名必须与证书预期的主机名匹配，否则验证将失败并导致连接错误” web.archive.org 。通过确保配置一致，您就能解决此503错误并恢复系统的正常运行。

参考资料：

- Istio官方文档与论坛：用户案例分析了使用自定义DNS导致的503错误及解决方案 web.archive.org web.archive.org 。
- VictoriaMetrics文档：vmaalert 重试和日志优化特性的更新 docs.victoriametrics.com 。
- Istio错误代码说明：503 UC 表示上游连接意外终止 web.archive.org 。

