Chapter 10 - Practical Alerting

Practical Alerting from Time-Series Data

Written by Jamie Wilkinson
Edited by Kavita Guliani

> May the queries flow, and the pager stay silent.
>
> Traditional SRE blessing

Monitoring, the bottom layer of the *Hierarchy of Production Needs*, is fundamental to running a stable service. Monitoring enables service owners to make rational decisions about the impact of changes to the service, apply the scientific method to incident response, and of course ensure their reason for existence: to measure the service's alignment with business goals (see Monitoring Distributed Systems).

Regardless of whether or not a service enjoys SRE support, it should be run in a symbiotic relationship with its monitoring. But having been tasked with ultimate responsibility for Google Production, SREs develop a particularly intimate knowledge of the monitoring infrastructure that supports their service.

Monitoring a very large system is challenging for a couple of reasons:

- The sheer number of components being analyze
- The need to maintain a reasonably low maintenance burden on the engineers responsible for the system

Google's monitoring systems don't just measure simple metrics, such as the average response time of an unladen European web server; we also need to understand the distribution of those response times across all web servers in that region. This knowledge enables us to identify the factors contributing to the latency tail.

At the scale our systems operate, being alerted for single-machine failures is unacceptable because such data is too noisy to be actionable. Instead we try to build systems that are robust against failures in the systems they depend on. Rather than requiring management of many individual components, a large system should be designed to aggregate signals and prune outliers. We need monitoring systems that allow us to alert for high-level service objectives, but retain the granularity to inspect individual components as needed.

Google's monitoring systems evolved over the course of 10 years from the traditional model of custom scripts that check responses and alert, wholly separated from visual display of trends, to a new paradigm. This new model made the collection of time-series a first-class role of the monitoring system, and replaced those check scripts with a rich language for manipulating time-series into charts and alerts.

The Rise of Borgmon

Shortly after the job scheduling infrastructure Borg [Ver15] was created in 2003, a new monitoring system —Borgmon—was built to complement it.

Time-Series Monitoring Outside of Google

This chapter describes the architecture and programming interface of an internal monitoring tool that was foundational for the growth and reliability of Google for almost 10 years…but how does that help you, our dear reader?

In recent years, monitoring has undergone a Cambrian Explosion: Riemann, Heka, Bosun, and Prometheus

have emerged as open source tools that are very similar to Borgmon's time-series–based alerting. In particular, Prometheus[42] shares many similarities with Borgmon, especially when you compare the two rule languages. The principles of variable collection and rule evaluation remain the same across all these tools and provide an environment with which you can experiment, and hopefully launch into production, the ideas inspired by this chapter.

Instead of executing custom scripts to detect system failures, Borgmon relies on a common data exposition format; this enables mass data collection with low overheads and avoids the costs of subprocess execution and network connection setup. We call this *white-box monitoring* (see Monitoring Distributed Systems for a comparison of white-box and black-box monitoring).

The data is used both for rendering charts and creating alerts, which are accomplished using simple arithmetic. Because collection is no longer in a short-lived process, the history of the collected data can be used for that alert computation as well.

These features help to meet the goal of simplicity described in Monitoring Distributed Systems. They allow the system overhead to be kept low so that the people running the services can remain agile and respond to continuous change in the system as it grows.

To facilitate mass collection, the metrics format had to be standardized. An older method of exporting the internal state (known as *varz*)[43] was formalized to allow the collection of all metrics from a single target in one HTTP fetch. For example, to view a page of metrics manually, you could use the following command:

```
% curl http://webserver:80/varz http_requests 37
errors_total 12
```

A Borgmon can collect from other Borgmon,[44] so we can build hierarchies that follow the topology of the service, aggregating and summarizing information and discarding some strategically at each level. Typically, a team runs a single Borgmon per cluster, and a pair at the global level. Some very large services shard below the cluster level into many *scraper* Borgmon, which in turn feed to the cluster-level Borgmon.

Instrumentation of Applications

The `/varz` HTTP handler simply lists all the exported variables in plain text, as space-separated keys and values, one per line. A later extension added a mapped variable, which allows the exporter to define several labels on a variable name, and then export a table of values or a histogram. An example *map-valued* variable looks like the following, showing 25 HTTP 200 responses and 12 HTTP 500s:

```
http_responses map:code 200:25 404:0 500:12
```

Adding a metric to a program only requires a single declaration in the code where the metric is needed.

In hindsight, it's apparent that this schemaless textual interface makes the barrier to adding new instrumentation very low, which is a positive for both the software engineering and SRE teams. However, this has a trade-off against ongoing maintenance; the decoupling of the variable definition from its use in Borgmon rules requires careful change management. In practice, this trade-off has been satisfactory because tools to validate and generate rules have been written as well.[45]

Exporting Variables

Google's web roots run deep: each of the major languages used at Google has an implementation of the exported variable interface that automagically registers with the HTTP server built into every Google

binary by default.[46] The instances of the variable to be exported allow the server author to perform obvious operations like adding an amount to the current value, setting a key to a specific value, and so forth. The Go `expvar` library[47] and its JSON output form have a variant of this API.

Collection of Exported Data

To find its targets, a Borgmon instance is configured with a list of targets using one of many name resolution methods.[48] The target list is often dynamic, so using service discovery reduces the cost of maintaining it and allows the monitoring to scale.

At predefined intervals, Borgmon fetches the `/varz` URI on each target, decodes the results, and stores the values in memory. Borgmon also spreads the collection from each instance in the target list over the whole interval, so that collection from each target is not in lockstep with its peers.

Borgmon also records "synthetic" variables for each target in order to identify:

- If the name was resolved to a host and port
- If the target responded to a collection
- If the target responded to a health check
- What time the collection finished

These synthetic variables make it easy to write rules to detect if the monitored tasks are unavailable.

It's interesting that varz is quite dissimilar to SNMP (Simple Network Management Protocol), which "is designed […] to have minimal transport requirements and to continue working when most other network applications fail" [Mic03]. Scraping targets over HTTP seems to be at odds with this design principle; however, experience shows that this is rarely an issue.[49] The system itself is already designed to be robust against network and machine failures, and Borgmon allows engineers to write smarter alerting rules by using the collection failure itself as a signal.

Storage in the Time-Series Arena

A service is typically made up of many binaries running as many tasks, on many machines, in many clusters. Borgmon needs to keep all that data organized, while allowing flexible querying and slicing of that data.

Borgmon stores all the data in an in-memory database, regularly checkpointed to disk. The data points have the form `(timestamp, value)`, and are stored in chronological lists called *time-series*, and each time-series is named by a unique set of *labels*, of the form `name=value`.

As presented in Figure 10-1, a time-series is conceptually a one-dimensional matrix of numbers, progressing through time. As you add permutations of labels to this time-series, the matrix becomes multidimensional.

Figure 10-1. A time-series for errors labeled by the original host each was collected from

In practice, the structure is a fixed-sized block of memory, known as the *time-series arena*, with a garbage collector that expires the oldest entries once the arena is full. The time interval between the most recent and oldest entries in the arena is the *horizon*, which indicates how much queryable data is kept in RAM. Typically, datacenter and global Borgmon are sized to hold about 12 hours of data[50] for rendering consoles, and much less time if they are the lowest-level collector shards. The memory requirement for a

single data point is about 24 bytes, so we can fit 1 million unique time-series for 12 hours at 1-minute intervals in under 17 GB of RAM.

Periodically, the in-memory state is archived to an external system known as the Time-Series Database (TSDB). Borgmon can query TSDB for older data and, while slower, TSDB is cheaper and larger than a Borgmon's RAM.

Labels and Vectors

As shown in the example time-series in Figure 10-2, time-series are stored as sequences of numbers and timestamps, which are referred to as *vectors*. Like vectors in linear algebra, these vectors are slices and cross-sections of the multidimensional matrix of data points in the arena. Conceptually the timestamps can be ignored, because the values are inserted in the vector at regular intervals in time—for example, 1 or 10 seconds or 1 minute apart.

Figure 10-2. An example time-series

The name of a time-series is a *labelset*, because it's implemented as a set of labels expressed as `key=value` pairs. One of these labels is the variable name itself, the key that appears on the varz page.

A few label names are declared as important. For the time-series in the time-series database to be identifiable, it must at minimum have the following labels:

`var`

: The name of the variable

`job`

: The name given to the type of server being monitored

`service`

: A loosely defined collection of jobs that provide a service to users, either internal or external

`zone`

: A Google convention that refers to the location (typically the datacenter) of the Borgmon that performed the collection of this variable

Together, these variables appear something like the following, called the *variable expression*:

```
{var=http_requests,job=webserver,instance=host0:80,service=web,zone=us-west}
```

A query for a time-series does not require specification of all these labels, and a search for a *labelset* returns all matching time-series in a vector. So we could return a vector of results by removing the `instance` label in the preceding query, if there were more than one instance in the cluster. For example:

```
{var=http_requests,job=webserver,service=web,zone=us-west}
```

might have a result of five rows in a vector, with the most recent value in the time-series like so:

```
{var=http_requests,job=webserver,instance=host0:80,service=web,zone=us-west} 10
{var=http_requests,job=webserver,instance=host1:80,service=web,zone=us-west} 9
{var=http_requests,job=webserver,instance=host2:80,service=web,zone=us-west} 11
{var=http_requests,job=webserver,instance=host3:80,service=web,zone=us-west} 0
```

```
{var=http_requests,job=webserver,instance=host4:80,service=web,zone=us-west} 10
```

Labels can be added to a time-series from:

- The target's name, e.g., the job and instance
- The target itself, e.g., via map-valued variables
- The Borgmon configuration, e.g., annotations about location or relabeling
- The Borgmon rules being evaluated

We can also query time-series in time, by specifying a duration to the variable expression:

```
{var=http_requests,job=webserver,service=web,zone=us-west}[10m]
```

This returns the last 10 minutes of history of the time-series that matched the expression. If we were collecting data points once per minute, we would expect to return 10 data points in a 10-minute window, like so:[51]

```
{var=http_requests,job=webserver,instance=host0:80, ...} 0 1 2 3 4 5 6 7 8 9 10
{var=http_requests,job=webserver,instance=host1:80, ...} 0 1 2 3 4 4 5 6 7 8 9
{var=http_requests,job=webserver,instance=host2:80, ...} 0 1 2 3 5 6 7 8 9 9 11
{var=http_requests,job=webserver,instance=host3:80, ...} 0 0 0 0 0 0 0 0 0 0 0
{var=http_requests,job=webserver,instance=host4:80, ...} 0 1 2 3 4 5 6 7 8 9 10
```

Rule Evaluation

Borgmon is really just a programmable calculator, with some syntactic sugar that enables it to generate alerts. The data collection and storage components already described are just necessary evils to make that programmable calculator ultimately fit for purpose here as a monitoring system. :)

Note

Centralizing the rule evaluation in a monitoring system, rather than delegating it to forked subprocesses, means that computations can run in parallel against many similar targets. This practice keeps the configuration relatively small in size (for example, by removing duplication of code) yet more powerful through its expressiveness.

The Borgmon program code, also known as *Borgmon rules*, consists of simple algebraic expressions that compute time-series from other time-series. These rules can be quite powerful because they can query the history of a single time-series (i.e., the time axis), query different subsets of labels from many time-series at once (i.e., the space axis), and apply many mathematical operations.

Rules run in a parallel threadpool where possible, but are dependent on ordering when using previously defined rules as input. The size of the vectors returned by their query expressions also determines the overall runtime of a rule. Thus, it is typically the case that one can add CPU resources to a Borgmon task in response to it running slow. To assist more detailed analysis, internal metrics on the runtime of rules are exported for performance debugging and for monitoring the monitoring.

Aggregation is the cornerstone of rule evaluation in a distributed environment. Aggregation entails taking the sum of a set of time-series from the tasks in a job in order to treat the job as a whole. From those sums, overall rates can be computed. For example, the total queries-per-second rate of a job in a datacenter is the sum of all the rates of change[52] of all the query counters.[53]

Tip

A counter is any monotonically non-decreasing variable—which is to say, counters only increase in value. Gauges, on the other hand, may take any value they like. Counters measure increasing values, such as the total number of kilometers driven, while gauges show current state, such as the amount of fuel remaining or current speed. When collecting Borgmon-style data, it's better to use counters, because they don't lose meaning when events occur between sampling intervals. Should any activity or changes occur between sampling intervals, a gauge collection is likely to miss that activity.

For an example web server, we might want to alert when our web server cluster starts to serve more errors as a percent of requests than we think is normal—or more technically, when the sum of the rates of non-HTTP-200 return codes on all tasks in the cluster, divided by the sum of the rates of requests to all tasks in that cluster, is greater than some value.

This is accomplished by:

1. Aggregating the rates of response codes across all tasks, outputting a vector of rates at that point in time, one for each code.
2. Computing the total error rate as the sum of that vector, outputting a single value for the cluster at that point in time. This total error rate excludes the 200 code from the sum, because it is not an error.
3. Computing the cluster-wide ratio of errors to requests, dividing the total error rate by the rate of requests that arrived, and again outputting a single value for the cluster at that point in time.

Each of these outputs at a point in time gets appended to its named variable expression, which creates the new time-series. As a result, we will be able to inspect the history of error rates and error ratios some other time.

The rate of requests rules would be written in Borgmon's rule language as the following:

```
rules <<<
      # Compute the rate of requests for each task from the count of requests
      {var=task:http_requests:rate10m,job=webserver} =
        rate({var=http_requests,job=webserver}[10m]);

      # Sum the rates to get the aggregate rate of queries for the cluster;
      # 'without instance' instructs Borgmon to remove the instance label
      # from the right hand side.
      {var=dc:http_requests:rate10m,job=webserver} =
        sum without instance({var=task:http_requests:rate10m,job=webserver})
    >>>
```

The `rate()` function takes the enclosed expression and returns the total delta divided by the total time between the earliest and latest values.

With the example time-series data from the query before, the results for the `task:http_requests:rate10m` rule would look like:[54]

```
      {var=task:http_requests:rate10m,job=webserver,instance=host0:80, ...} 1
      {var=task:http_requests:rate10m,job=webserver,instance=host1:80, ...} 0.9
      {var=task:http_requests:rate10m,job=webserver,instance=host2:80, ...} 1.1
      {var=task:http_requests:rate10m,job=webserver,instance=host3:80, ...} 0
      {var=task:http_requests:rate10m,job=webserver,instance=host4:80, ...} 1
```

and the results for the `dc:http_requests:rate10m` rule would be:

```
      {var=dc:http_requests:rate10m,job=webserver,service=web,zone=us-west} 4
```

because the second rule uses the first one as input.

Note

The `instance` label is missing in the output now, discarded by the aggregation rule. If it had remained in the rule, then Borgmon would not have been able to sum the five rows together.

In these examples, we use a time window because we're dealing with discrete points in the time-series, as opposed to continuous functions. Doing so makes the rate calculation easier than performing calculus, but means that to compute a rate, we need to select a sufficient number of data points. We also have to deal with the possibility that some recent collections have failed. Recall that the historical variable expression notation uses the range `[10m]` to avoid missing data points caused by collection errors.

The example also uses a Google convention that helps readability. Each computed variable name contains a colon-separated triplet indicating the aggregation level, the variable name, and the operation that created that name. In this example, the lefthand variables are "task HTTP requests 10-minute rate" and "datacenter HTTP requests 10-minute rate."

Now that we know how to create a rate of queries, we can build on that to also compute a rate of errors, and then we can calculate the ratio of responses to requests to understand how much useful work the service is doing. We can compare the ratio rate of errors to our service level objective (see Service Level Objectives) and alert if this objective is missed or in danger of being missed:

```
rules <<<
      # Compute a rate pertask and per 'code' label
      {var=task:http_responses:rate10m,job=webserver} =
        rate by code({var=http_responses,job=webserver}[10m]);

      # Compute a cluster level response rate per 'code' label
      {var=dc:http_responses:rate10m,job=webserver} =
        sum without instance({var=task:http_responses:rate10m,job=webserver});

      # Compute a new cluster level rate summing all non 200 codes
      {var=dc:http_errors:rate10m,job=webserver} =
        sum without code({var=dc:http_responses:rate10m,job=webserver,code=!/200/});

      # Compute the ratio of the rate of errors to the rate of requests
      {var=dc:http_errors:ratio_rate10m,job=webserver} =
        {var=dc:http_errors:rate10m,job=webserver}
          /
        {var=dc:http_requests:rate10m,job=webserver};
    >>>
```

Again, this calculation demonstrates the convention of suffixing the new time-series variable name with the operation that created it. This result is read as "datacenter HTTP errors 10 minute ratio of rates."

The output of these rules might look like:[55]

```
{var=task:http_responses:rate10m,job=webserver}

        {var=task:http_responses:rate10m,job=webserver,code=200,instance=host0:80, ...} 1
      {var=task:http_responses:rate10m,job=webserver,code=500,instance=host0:80, ...} 0
      {var=task:http_responses:rate10m,job=webserver,code=200,instance=host1:80, ...} 0.5
      {var=task:http_responses:rate10m,job=webserver,code=500,instance=host1:80, ...} 0.4
      {var=task:http_responses:rate10m,job=webserver,code=200,instance=host2:80, ...} 1
      {var=task:http_responses:rate10m,job=webserver,code=500,instance=host2:80, ...} 0.1
      {var=task:http_responses:rate10m,job=webserver,code=200,instance=host3:80, ...} 0
      {var=task:http_responses:rate10m,job=webserver,code=500,instance=host3:80, ...} 0
      {var=task:http_responses:rate10m,job=webserver,code=200,instance=host4:80, ...} 0.9
      {var=task:http_responses:rate10m,job=webserver,code=500,instance=host4:80, ...} 0.1


{var=dc:http_responses:rate10m,job=webserver}

      {var=dc:http_responses:rate10m,job=webserver,code=200, ...} 3.4
      {var=dc:http_responses:rate10m,job=webserver,code=500, ...} 0.6
```

```
{var=dc:http_responses:rate10m,job=webserver,code=!/200/}

      {var=dc:http_responses:rate10m,job=webserver,code=500, ...} 0.6

{var=dc:http_errors:rate10m,job=webserver}

      {var=dc:http_errors:rate10m,job=webserver, ...} 0.6

{var=dc:http_errors:ratio_rate10m,job=webserver}

      {var=dc:http_errors:ratio_rate10m,job=webserver} 0.15
```

Note

The preceding output shows the intermediate query in the `dc:http_errors:rate10m` rule that filters the non-200 error codes. Though the value of the expressions are the same, observe that the code label is retained in one but removed from the other.

As mentioned previously, Borgmon rules create new time-series, so the results of the computations are kept in the time-series arena and can be inspected just as the source time-series are. The ability to do so allows for ad hoc querying, evaluation, and exploration as tables or charts. This is a useful feature for debugging while on-call, and if these ad hoc queries prove useful, they can be made permanent visualizations on a service console.

Alerting

When an alerting rule is evaluated by a Borgmon, the result is either true, in which case the alert is triggered, or false. Experience shows that alerts can "flap" (toggle their state quickly); therefore, the rules allow a minimum duration for which the alerting rule must be true before the alert is sent. Typically, this duration is set to at least two rule evaluation cycles to ensure no missed collections cause a false alert.

The following example creates an alert when the error ratio over 10 minutes exceeds 1% and the total number of errors exceeds 1 per second

```
rules <<<
      {var=dc:http_errors:ratio_rate10m,job=webserver} > 0.01
        and by job, error
      {var=dc:http_errors:rate10m,job=webserver} > 1
        for 2m
        => ErrorRatioTooHigh
          details "webserver error ratio at %trigger_value%"
          labels { severity=page };
    >>>
```

Our example holds the ratio rate at 0.15, which is well over the threshold of 0.01 in the alerting rule. However, the number of errors is not greater than 1 at this moment, so the alert won’t be active. Once the number of errors exceeds 1, the alert will go *pending* for two minutes to ensure it isn’t a transient state, and only then will it *fire*.

The alert rule contains a small template for filling out a message containing contextual information: which job the alert is for, the name of the alert, the numerical value of the triggering rule, and so on. The contextual information is filled out by Borgmon when the alert fires and is sent in the Alert RPC.

Borgmon is connected to a centrally run service, known as the Alertmanager, which receives Alert RPCs when the rule first triggers, and then again when the alert is considered to be "firing." The Alertmanager is responsible for routing the alert notification to the correct destination. Alertmanager can be configured to

do the following:

- Inhibit certain alerts when others are active
- Deduplicate alerts from multiple Borgmon that have the same labelsets
- Fan-in or fan-out alerts based on their labelsets when multiple alerts with similar labelsets fire

As described in Monitoring Distributed Systems, teams send their page-worthy alerts to their on-call rotation and their important but subcritical alerts to their ticket queues. All other alerts should be retained as informational data for status dashboards.

A more comprehensive guide to alert design can be found in Service Level Objectives.

Sharding the Monitoring Topology

A Borgmon can import time-series data from other Borgmon, as well. While one could attempt to collect from all tasks in a service globally, doing so quickly becomes a scaling bottleneck and introduces a single point of failure into the design. Instead, a streaming protocol is used to transmit time-series data between Borgmon, saving CPU time and network bytes compared to the text-based varz format. A typical such deployment uses two or more global Borgmon for top-level aggregation and one Borgmon in each datacenter to monitor all the jobs running at that location. (Google divides the production network into zones for production changes, so having two or more global replicas provides diversity in the face of maintenance and outages for this otherwise single point of failure.)

As shown in Figure 10-3, more complicated deployments shard the datacenter Borgmon further into a purely scraping-only layer (often due to RAM and CPU constraints in a single Borgmon for very large services) and a DC aggregation layer that performs mostly rule evaluation for aggregation. Sometimes the global layer is split between rule evaluation and dashboarding. Upper-tier Borgmon can filter the data they want to stream from the lower-tier Borgmon, so that the global Borgmon does not fill its arena with all the per-task time-series from the lower tiers. Thus, the aggregation hierarchy builds local caches of relevant time-series that can be drilled down into when required.

Figure 10-3. A data flow model of a hierarchy of Borgmon in three clusters

Black-Box Monitoring

Borgmon is a white-box monitoring system—it inspects the internal state of the target service, and the rules are written with knowledge of the internals in mind. The transparent nature of this model provides great power to identify quickly what components are failing, which queues are full, and where bottlenecks occur, both when responding to an incident and when testing a new feature deployment.

However, white-box monitoring does not provide a full picture of the system being monitored; relying solely upon white-box monitoring means that you aren't aware of what the users see. You only see the queries that arrive at the target; the queries that never make it due to a DNS error are invisible, while queries lost due to a server crash never make a sound. You can only alert on the failures that you expected.

Teams at Google solve this coverage issue with Prober, which runs a protocol check against a target and reports success or failure. The prober can send alerts directly to Alertmanager, or its own varz can be collected by a Borgmon. Prober can validate the response payload of the protocol (e.g., the HTML contents of an HTTP response) and validate that the contents are expected, and even extract and export values as time-series. Teams often use Prober to export histograms of response times by operation type and payload size so that they can slice and dice the user-visible performance. Prober is a hybrid of the check-

and-test model with some richer variable extraction to create time-series.

Prober can be pointed at either the frontend domain or behind the load balancer. By using both targets, we can detect localized failures and suppress alerts. For example, we might monitor both the load balanced *www.google.com* and the web servers in each datacenter behind the load balancer. This setup allows us to either know that traffic is still served when a datacenter fails, or to quickly isolate an edge in the traffic flow graph where a failure has occurred.

Maintaining the Configuration

Borgmon configuration separates the definition of the rules from the targets being monitored. This means the same sets of rules can be applied to many targets at once, instead of writing nearly identical configuration over and over. This separation of concerns might seem incidental, but it greatly reduces the cost of maintaining the monitoring by avoiding lots of repetition in describing the target systems.

Borgmon also supports language templates. This macro-like system enables engineers to construct libraries of rules that can be reused. This functionality again reduces repetition, thus reducing the likelihood of bugs in the configuration.

Of course, any high-level programming environment creates the opportunity for complexity, so Borgmon provides a way to build extensive unit and regression tests by synthesizing time-series data, in order to ensure that the rules behave as the author thinks they do. The Production Monitoring team runs a continuous integration service that executes a suite of these tests, packages the configuration, and ships the configuration to all the Borgmon in production, which then validate the configuration before accepting it.

In the vast library of common templates that have been created, two classes of monitoring configuration have emerged. The first class simply codifies the emergent schema of variables exported from a given library of code, such that any user of the library can reuse the template of its varz. Such templates exist for the HTTP server library, memory allocation, the storage client library, and generic RPC services, among others. (While the varz interface declares no schema, the rule library associated with the code library ends up declaring a schema.)

The second class of library emerged as we built templates to manage the aggregation of data from a single-server task to the global service footprint. These libraries contain generic aggregation rules for exported variables that engineers can use to model the topology of their service.

For example, a service may provide a single global API, but be homed in many datacenters. Within each datacenter, the service is composed of several shards, and each shard is composed of several jobs with various numbers of tasks. An engineer can model this breakdown with Borgmon rules so that when debugging, subcomponents can be isolated from the rest of the system. These groupings typically follow the shared fate of components; e.g., individual tasks share fate due to configuration files, jobs in a shard share fate because they’re homed in the same datacenter, and physical sites share fate due to networking.

Labeling conventions make such division possible: a Borgmon adds labels indicating the target’s instance name and the shard and datacenter it occupies, which can be used to group and aggregate those time-series together.

Thus, we have multiple uses for labels on a time-series, though all are interchangeable:

- Labels that define breakdowns of the data itself (e.g., our HTTP response code on the `http_responses` variable
- Labels that define the source of the data (e.g., the instance or job name)
- Labels that indicate the locality or aggregation of the data within the service as a whole (e.g., the zone label describing a physical location, a shard label describing a logical grouping of tasks)

The templated nature of these libraries allows flexibility in their use. The same template can be used to aggregate from each tier.

Ten Years On…

Borgmon transposed the model of check-and-alert per target into mass variable collection and a centralized rule evaluation across the time-series for alerting and diagnostics.

This decoupling allows the size of the system being monitored to scale independently of the size of alerting rules. These rules cost less to maintain because they're abstracted over a common time-series format. New applications come ready with metric exports in all components and libraries to which they link, and well-traveled aggregation and console templates, which further reduces the burden of implementation.

Ensuring that the cost of maintenance scales sublinearly with the size of the service is key to making monitoring (and all sustaining operations work) maintainable. This theme recurs in all SRE work, as SREs work to scale all aspects of their work to the global scale.

Ten years is a long time, though, and of course today the shape of the monitoring landscape within Google has evolved with experiments and changes, striving for continual improvement as the company grows.

Even though Borgmon remains internal to Google, the idea of treating time-series data as a data source for generating alerts is now accessible to everyone through those open source tools like Prometheus, Riemann, Heka, and Bosun, and probably others by the time you read this.

[42]Prometheus is an open source monitoring and time-series database system available at *http://prometheus.io*.

[43]Google was born in the USA, so we pronounce this *"var-zee."*

[44]The plural of Borgmon is Borgmon, like sheep.

[45]Many non-SRE teams use a generator to stamp out the initial boilerplate and ongoing updates, and find the generator much easier to use (though less powerful) than directly editing the rules.

[46]Many other applications use their service protocol to export their internal state, as well. OpenLDAP exports it through the `cn=Monitor` subtree; MySQL can report state with a `SHOW VARIABLES` query; Apache has its `mod_status` handler.

[47]*https://golang.org/pkg/expvar/*

[48]The Borg Name System (BNS) is described in The Production Environment at Google, from the Viewpoint of an SRE.

[49]Recall in Monitoring Distributed Systems the distinction between alerting on symptoms and on causes.

[50]This 12-hour horizon is a magic number that aims to have enough information for debugging an incident in RAM for fast queries without costing *too much* RAM.

[51]The `service` and `zone` labels are elided here for space, but are present in the returned expression.

[52]Computing the sum of rates instead of the rate of sums defends the result against counter resets or missing data, perhaps due to a task restart or failed collection of data.

[53]Despite being untyped, the majority of varz are simple counters. Borgmon's rate function handles all the corner cases of counter resets.

[54]The `service` and `zone` labels are elided for space.

[55]The `service` and `zone` labels are elided for space.

previous

Part III - Practices

next

Chapter 11 - Being On-Call