O'REILLY®

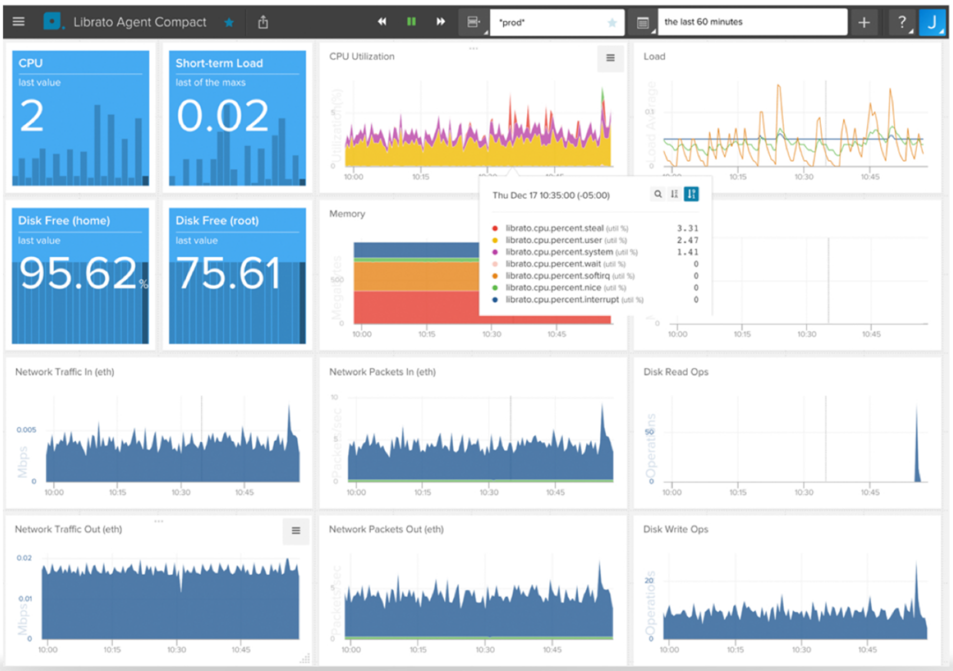
Compliments of librato®

Monitoring Taxonomy

Laying Out the Tools Landscape

Dave Josephsen

librato®



Thu Dec 17 10:35:00 (-05:00)

```
● librato.cpu.percent.steal (util %)      3.31
● librato.cpu.percent.user (util %)       2.47
● librato.cpu.percent.system (util %)     1.41
● librato.cpu.percent.wait (util %)          0
● librato.cpu.percent.softirq (util %)       0
● librato.cpu.percent.nice (util %)          0
● librato.cpu.percent.interrupt (util %)     0
```

GO FROM ZERO TO FULL-STACK MONITORING IN MINUTES

Use the open-source tools you love while offloading your metrics and alerting to a team you can trust.

Sign up for a free trial today: librato.com/freetrial

Monitoring Taxonomy

Laying Out the Tools Landscape

Dave Josephsen

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Monitoring Taxonomy
by Dave Josephsen

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Editor: Virginia Wilson
Production Editor: Colleen Cole
Copyeditor: Octal Publishing, Inc.
Interior Designer: David Futato
Cover Designer: Karen Montgomery

November 2016: First Edition

Revision History for the First Edition
2016-11-08: First Release

978-1-491-95081-4

Table of Contents

CHAPTER 1

Welcome! Read This First

There are few things you need to know before we begin.

First, what you're holding in your hand is not the entire report. This print version only includes the 25 open source tools from the full taxonomy. Think of it as *the teaser*. The rest (62 tools in all) are online at *http://github.com/librato/taxonomy*.

Second, this report was not intended to be read cover-to-cover. This is a reference work. What I'm trying to do here is categorize your problem and then provide you some descriptions of monitoring tools that might help you solve your problem. I'm not trying to fully document every monitoring tool, I'm just trying to get you pointed in the right direction. See the following section for a description of how we're going to do that.

Third, I didn't choose the tools that were included in this report. They are self-selected based on their use in the real world by other engineers like you. See "How Did You Choose the Tools?" on page 5 if you're curious about why tool X was included but not tool Y.

How Does This Report Work?

This report is predicated on the assumption that you hold in your mind a loose set of requirements (or at least desires) for a monitoring system. Maybe you need an open source tool that can inspect SFlow traffic and alert you to a bit-torrent happening on your network, or maybe you need a commercial byte-code injection tool that can measure latency between your worker threads and your Kafka queue.

Maybe you don't know exactly what you're requirements are, but you'd like to weigh the options either way, or maybe you're aware of one popular tool and want to learn about other tools that do the same thing.

This report tries to help you out in any of those cases by categorizing your problem and then presenting you with a group of tools that might work for you. We use a hierarchy in which you begin by choosing how you want to manage the tool (hosted? on-premises?), how you want to pay for it, and so on, filtering out the tools that don't apply to you as you proceed.

In this document, I categorize and describe 25 different open source monitoring tools along a path of four taxa. Then, I provide a summary of each tool in the form of a small questionnaire to give you a sense of how the tool does what it does, how well it gets along with other tools, and how well it might fit into your environment/culture/stack.

Let's Begin

There are four top-level categories, and each have a varying number of classification, which we define in the sections that follow.

Operations Burden

Who do you want to run your monitoring tool? The classifications are as folows:

Traditional
: You download and install tools in this category on your network.

Hosted
: Another party runs the tools in this category for you, off site.

Appliances and Sensors
: Tools in this category encompass vendor-provided hardware that is either managed by the vendor or you, and other hybrid models that involve hardware.

Pay Model

How do you want to pay for your monitoring tool? Here are the classifications:

Free/open
: Open source tools cost nothing to obtain and come with their source code.

Commercial
: Commercially licensed software often costs money to obtain (legally) and is usually distributed in binary form. This category includes demo software that's free to use for a limited time as well as tools whose free-to-use tiers are too limited on which to run a small startup.

Freemium
: Freemium software is free to use but comes with a paid premium component or usage tier. You can find commercial tools that have a usable free tier. By usable we mean the free tier provides the base-line operability a resonable person would consider using for running a small startup.

Free/Closed
: This is closed source software that doesn't cost money to obtain. You can find shareware and nagware tools here.

Subscription hardware
: Appliances in this category typically provide the hardware for free but charge a monthly or annual subscription fee for use.

Activity Model

Do you need a tool to collect measurements or process measurements, or both? Here are the classifications to look for:

Collectors
: These are tools designed primarily to measure things and make observations. This includes monitoring agents, instrumentation libraries, and sensors.

Processors
: These are tools designed primarily to accept and/or process data from other tools. This includes data visualization tools and time

series databases as well as stream processing systems and glue projects.

Monoliths
: This category of tools is designed to be all-in-one solutions. It's probably possible to import/export data from these tools, but they were designed to consume the data that they collect themselves. Most traditional operations-oriented "monitoring" tools (e.g., openview, patrol, and Nagios) fit into this category.

Focus

What do you need your tool to actually monitor? Here are the classifications:

System availability
: These tools seek to answer the question "Is it up?" You will find any tool that was primarily designed to check for system or service availability at a one-minute or greater resolution. Most classic operations-centric monitoring tools like Nagios fall into this category.

App/database performance
: Application performance management (APM) tools insert themselves into popular databases and language interpretors for the purpose of analyzing their performance. This is usually done by patching the interpretor or other binary with instrumentation code. These tools can give very detailed performance data at a highly granular resolution on databases like MySQL or even on custom apps by, for example, instrumenting the java virtual machine (JVM). Examples include New Relic, Appneta, and Vivid Cortex.

Networks
: This is a broad class of tools designed to monitor and analyze network availability, performance, and content. Packet taps, SNMP collectors, Netflow, and SFlow related tools can be found herein.

Data Processing
: Tools in this category were designed to collect or accept ad hoc metrics and log data with the intention to do something useful with it such as visualizing it (drawing graphs), parsing it, transforming it, alerting on it, and possibly forwarding it to other

tools. Examples include Splunk, Heka, Graphite, Riemann, and Librato.

Storage Analysis
: These tools specialize in analyzing storage area networks (SAN) and other distributed storage technologies.

Platform-Specific
: Tools in this category were created specifically to monitor a particular platform, or a class of platforms. AWS Cloudwatch, Heroku metrics, and Rackspace monitoring all fall into this category

Ok, I Think I Know What I'm Looking For, Now What?

Each chapter in this report (apart from this one) represents one taxonomic path. So, if you were looking for a traditional → free/open → monolithic → system_availability–type tool—which happens to be the most popular taxonomic path (Nagios and its myriad clones [12 tools in all] categorize there)—you would thumb to Chapter 5.

How Did You Choose the Tools?

The initial list of tools I'm documenting were taken from James Turnbull's "2015 Monitoring Survey" results, which you can see for yourself at *https://kartar.net/2015/08/monitoring-survey-2015---tools*. I'll be expanding on that, and I'm also happy to take pull requests for new tools to include or corrections to anything I've misrepresented.

Again, in this short document you'll find only *open source* tools that have a minimum of two respondents in the Katar survey. You can find the rest of the tools (commercial, open source, and points in between—62 in all) at the gitHub repo for this project at *http://github.com/librato/taxonomy*.

Why Did You Write This?

I wrote this because monitoring is a mess.

What does it even mean!? *Monitoring*…I mean, what do you want to know? Do you want to know that the servers are up? Do you want to know the maximum of the p95 request latency on your frontend

API? Do you want to know that someone left the back door open in your Albuquerque datacenter? These are very different problems. Each requires a different set of tools and expertise, and yet there's no way to categorically research any of these tools independently of the others. They all somehow belong to the same category—monitoring. I find this confounding in the extreme—the magnitude of our hand-wavey imprecision in describing this discipline. Especially in the context of our wholesale obsession with definition. Every other engineering discipline in the world is endlessly subcategorizing itself into oblivion with increasingly incomprehensible (but extremely precise) jargon. But nope: all of these tools are just "monitoring tools" (he says, waving his hand in the general direction of a massive assortment of completely different tools).

When science has a vast and incomprehensible landscape to make sense of, it creates a taxonomy, and that's what I'd like to begin here. I want to direct the attention of overwhelmed engineers toward the subset of "monitoring" tools that are likely to work for their problem domain in their corporate culture. Like a map; something to augment your already well-developed sense of direction and help you to avoid stepping in anything too smelly.

CHAPTER 2

Monitoring

A Few Types of Monitoring Systems

Throughout the taxonomy, I'll refer to this or that tool being a "centralized poller" or an "APM" tool, and on, so let's get some jargon straight before we move ahead.

Centralized Pollers

The oldest tools we describe as "monitoring tools" are centralized pollers. They are usually single hosts that sit somewhere on the network and, like the unfortunate employee in those cell-provider commercials, constantly asks the question "can you hear me now?" to every host you configure it to "monitor." If any of those hosts fail to respond, the system sends you a notification (typically via SMS, but more commonly these days via an alerting service like PagerDuty or VictorOps).

Strict centralized pollers are notoriously difficult to scale and configure. They also typically work on a one-minute or greater "tick," which limits their resolution potential in the context of monitoring for performance versus availability.

Passive Collectors

Most modern monitoring systems sit passively on the network and wait for updates from remote agents. Passive collectors always employ agents, which enables them to scale better than centralized pollers. Many systems also toe the line, using both active polling and passive collection as needs require.

Roll-Up Collectors

Roll-up patterns are used often in the high performance computing world where we find thousands of individual on-premises hosts running CPU or memory-bound processes. The idea is to use a series of systematic roll-ups between systems to minimize the polling load. Imagine a rack of servers that all summarize and send their metrics to the server at the top of their rack).

Roll-up systems can collect metrics on the order of seconds from a massive number of individual hosts. A good example of this pattern in the wild is Ganglia, whose Gmond agent uses a multicast gossip protocol to detect and aggregate metrics between systems in the same cluster.

Process Emitters/Reporters

In this report, when I use the word "instrumentation," I'm describing code that resides within a running process that you want to monitor. Instrumentation libraries enable software developers to take measurements from within their applications, counting the number of times this or that function is invoked and timing interactions with services like databases and external APIs.

Instrumentation libraries typically use one of two patterns internally. *Emitters* immediately purge their metrics via a nonblocking channel (usually interprocess or UDP to a locally listening socket), whereas *Reporters* use a nonblocking thread to hold their metrics in-memory, reporting them only when asked (usually via a listening network socket. StatsD for example, is a wildly popular process-emitter; Dropwizard metrics is a great process-reporter.

Application Performance Monitoring

Application Performance Monitoring (APM) tries to measure the performance characteristics of a web application from one end to the other; breaking down how long every little hunk of code took to do its thing. Thus, when someone says "the website is slow" you can hopefully see where you need to go to fix it. They are the current darlings of the “monitoringosphere,” and also currently contain the highest concentration of snake-oil.

Typically these tools use byte-code injection and/or monkey-patching to modify your code, compiler, or interpretor at runtime, wrapping the built-in classes and functions with modified versions that extract timing information. Those timing numbers are then emitted as metrics and sent into the APM’s data collection framework. Data resolution and retention varies widely, though many of these tools work on a 60-second tick, which is perfectly adequate in real life.

Real User Monitoring

Real User Monitoring (RUM) used to be a thing before it was swallowed by APM. The idea is that because you have JavaScript running within your user’s browser, you might as well inject a timer or two, just to see how long it took that user to load your page. RUM is probably the biggest deal in web-performance monitoring in the last decade or so. It’s a great way to get a real idea of what your user-experience looks like from your user’s perspective.

Although I mention RUM here and there in the taxonomy, there aren’t very many RUM-only tools out there (none are represented in the taxonomy currently). This is largely because RUM has become a feature of the various APM systems. Bucky (*http://github.hubspot.com/bucky*) is a good open source library if you’re just looking for some standalone RUM.

Exception Tracking

Exception tracking systems count crashes and panics. (In polite society we refer to them as “exceptions.”) They usually do this from within web-based applications server-side, or in code running in your user’s browser on the client side. They often walk the line between RUM and log-processing systems, having neither the open-endedness of logging systems nor the performance-centric focus of RUM. Their simplicity and single-mindedness make them cheap, easy to configure, simple to use, and very reliable. A collection of happy factors that combine to often make them the go-to application monitoring solution when centralized logging is too much of a hassle, and APM/RUM is too confusing, unreliable and/or expensive.

Remote Polling

Remote pollers use a geographically distributed array of "sensors" (machines capable of running `/bin/ping` [I jest, they also run telnet]) to collect availability data and sometimes some light performance data from a public-facing service like your website or mail server. Many APM tools and commercial monoliths (like Circonus) have their own remote polling networks which they operate alongside their other offerings, but there are also plenty of commercial offerings that specialize in remote polling as a service (like Pingdom and Nodeping).

A Few Things That You Should Know About Monitoring Systems

While we're here, I might as well give you some unsolicited advice. Consider what follows to be just one person's extremely well-informed and objectively correct opinion.

Think Big, But Use Small Tools

New monitoring systems spring into being for myriad reasons. The best ones come from engineers like you who couldn't find the tool they needed to get their work done. These tools tend to be small, cooperative, and purpose-specific.

Other tools come from people who were looking for a market niche. These are large, monolithic, and sprawly. Monitoring tools excel when they begin with a very strong focus and iterate on it until it's rock-solid. You'll probably need to use more than one monitoring tool, and that's OK.

Push versus Pull

This is our analog to *Emacs versus Vi*, and like that debate, it was never very interesting to begin with. Pull-based tools need to work harder, so they don't scale as well (most are fine into the realm of 7,000 hosts or so). Push-based tools just sit around and wait for stuff to happen, so they're easier to scale and they can take measurements more often. Yep, that's pretty much it, there's no real reason to become hung up on it; each technique has its place. When you encounter a pull-based system, be sure to ask yourself if it meets your scalability and resolution requirements.

Agent versus Agentless

Another of the classic debates. In the ’90s, monitoring agents were awful, and certainly to be avoided. These days, however, they’re generally well behaved, and the systems that use them are to be preferred over agentless systems, which invariably seem to commit some egregious security no-no like running under domain-admin privileges or centrally storing general-purpose credentials to all your boxes. Beware of agentless systems, and make sure you understand the security trade-offs they make.

Data Summarization and Storage

Time series data storage is very difficult. Anyone who says otherwise is trying to sell you something. The two big problems are dealing with the accumulation of a constantly increasing dataset, and implementing fast queries on a write-optimized data store. To get around these problems, most time series databases (TSDB) automatically perform some form of data summarization (also referred to as aggregation or roll-ups). The goal is to consolidate the individual data points you collect into fewer data points that accurately summarize the original measurements.

This report doesn’t afford me the room to delve very deeply into this subject, but suffice it to say that monitoring systems vary widely in how well they implement data aggregation. That’s relevant because you want to ensure that your data isn’t destroyed in the summarization process. On one extreme are the systems that expose all of the dials and knobs to you, giving you all the rope you need to hang yourself, and walking away, whereas on the other are the systems that try to insulate you by blithely destroying your data for you and assuming you’re too stupid or apathetic to care.

Good metrics systems are clear about their data retention and resolution schedules. They’ll say “we store 1-second resolution data for *X* days, and then we aggregate it by doing *Y*.” Beware of systems that don’t provide this information. They’re probably averaging your data into oblivion without telling you. Many systems will make you classify your inbound metrics, assigning to them types like “gauge,” “counter,” or “timer.” These types almost always directly map to data-summarization schemes internally; for example, applying mean-average to consolidate *gauge* metrics. Make sure you understand what your system does with classifications like these.

Finally, several systems employ "data-point capping," which is something that leaves me with mixed feelings. That is, regardless of the underlying storage resolution, their UI will display only *X* data points per graph (usually a number in the range of 300–800). This limits the amount of raw-resolution data you're allowed to see at once (the system will automatically average your data as you stretch the X-axis). I totally understand the impulse behind this, but I often find it frustating in practice. Also, the cynic in me suspects systems that do this are choosing responsive UI over data resolution to protect their own help desk (or more likely their Twitter feed) from complaints about slow graphs. As an end user who is using the system in a problem-solving capacity, I feel that is a mistake I should be allowed make when I need to. Caveat Emptor.

Autodiscovery

Take heed of tools that tout host autodiscovery as a feature (they should be promoting their compatibility with Chef/Puppet/Ansible/SaltStack, instead). Network-scanning autodiscovery features inside monitoring systems (which honestly never worked very well anyway) are not helping. They're just enabling IT management incompetency. I'm sorry if that sounds harsh, I'm just telling you how it is because I love you.

Data-to-Ink Ratio

Data visualization is an entire field of study, and although I can't prove it, I strongly believe that that no discipline ignores its tenants as strenuously as the discipline of IT monitoring. I really would love to drone on for pages on this topic, but alas every word I write here is one less word I can write in Part II, so read up on data visualization and avoid monitoring systems with pie and doughnut charts as well as visualizations that use a lot of screen to depict not a lot of data (maps, speedometers, etc.).

CHAPTER 3
traditional.free_open.collectors.data

This is a lonely category. Most monitoring collectors focus on system metrics like CPU, disk, and memory, but the tools in this path are general-purpose data collectors.

StatsD: Simple Daemon for Stats Aggregation

StatsD is a cache-and-forward metrics summarization daemon. It listens for metrics on a network socket, holds/aggregates them for some amount of time (10 seconds by default), and then emits a summary of the data (percentiles, sums, averages, etc.) to one or more (pluggable) backend metrics services like Graphite or Librato.

StatsD is also a vast collection of clients for the StatsD server, which make it trivial to emit metrics into a listening StatsD interface from pretty much anything. These include command-line tools as well as integrations for other monitoring systems and language bindings for myriad programming languages. StatsD is pretty much the de facto means of wiring together a metrics-centric toolchain today.

Push, Pull, Both, or Neither?

StatsD is a push-based system.

Measurement Resolution

The standard tick is 10 seconds.

Data Storage

StatsD is in-memory cache-and-forward: there is no data-storage requirement.

Analysis Capabilities

None. StatsD is intended to be pointed at the analysis tool of your choice.

Notification Capabilities

None. Notification typically occurs before StatsD (at a monitoring system like Nagios) or after StatsD hands off the metrics (in a processing system like Riemann or Graphite/Bosun)

Integration Capabilities

StatsD integrates with everything from which or to which you might ever want to get a metric.

Scaling Model

StatsD is a small-footprint, stateless daemon with a very simple text-based wire protocol. It is common practice to run a StatsD on every host system you want to monitor and/or chain together multiple instances.

CHAPTER 4

traditional.free_open.collectors.system

You will find most standalone open source metrics collectors here. Right now only collectD is listed, but I expect this section will grow as other standalone collectors (like Diamond and Telegraf) become more popular.

CollectD: Everybody's Favorite Monitoring Agent

CollectD is a Unix daemon that collects, and transfers performance data from your hosts. You can think of it as a standalone open-source agent. You install it on every system you want to monitor. It collects metrics from your systems and emits them to other monitoring systems for processing.

CollectD was written in C and is extremely fast and modular. Many plug-ins exist that enable it to collect metrics on just about any off-the-shelf software you might be running from Apache to Zookeeper.

Push, Pull, Both, or Neither?

CollectD is a push-based system.

Measurement resolution

The CollectD daemon reports every 10 seconds by default.

Data Storage

None. Data acquisition and storage is handled by plug-ins.

Analysis Capabilities

None. CollectD is a standalone agent, designed to interact with other monitoring systems for analysis.

Notification Capabilities

You can configure notifications by setting a severity and a time. This is useful for informing users about a noticeable condition such as high CPU load. When a notification is triggered it is dispatched the same way that the performance metrics are.

Various plug-ins can send or receive notifications. For instance the LogFile plug-in will write notifications to the log file, whereas the Network plug-in sends and receives notifications.

Integration Capabilities

CollectD uses a modular design, and you can install it on Linux, Solaris, macOS, AIX, FreeBSD, NetBSD, OpenBSD, and OpenWrt-powered devices. Support for Microsoft Windows is provided by SSC Serv, a native Windows service that implements CollectD's network protocol.

The vast library of 90-plus plug-ins allows for additional integrations with tools and services, including various web servers, interpreters, databases, and applications.

Scaling Model

CollectD is stateless and will scale linearly with respect to your instances.

CHAPTER 5
traditional.free_open.monoliths.data

In this category, you can find general-purpose monitoring systems that rely on metrics. These systems typically employ an agent to collect data and send it to a daemon for aggregation and processing.

Consul: Not What You Probably Meant by "Monitoring"

Consul is mostly a service discovery tool, by which I mean it is a service you run that enables entities like web servers to find other entities like backend API endpoints and databases without the need for hardcoded IPs. It does this either via HTTP/GET requests or via its own DNS service. Generally speaking, you configure your thingy to ask for a node address that can service requests of type foo, and Consul responds with the address of the best node to which to send requests of that type.

Consul is, in fact, a very good and robust service discovery tool, by which I mean, it works quite hard, when queried for a service, to respond with the best possible node address in your infrastructure to handle requests of that type, and especially to avoid responding with the address of a node that is unreachable or overburdened. In the pursuit of that highly accurate response, Consul has its own internal monitoring system, which works with its own *gossip protocol* and enables you to configure ad hoc service checks (conceptually identical to Nagios checks, to include the return codes).

Therefore, although Consul is not a monitoring system, it does monitor the hosts for which it is responsible. And given its robust gossip and consensus protocols, does a better job than most dedicated monitoring systems of detecting host and service outages (usually detecting them on the order of 300–500 milliseconds). You can configure *node* checks or *service* checks. If any single node check fails, every service on the host is marked down, and that host's address will not appear in Consul service responses. If any service check fails, the corresponding service on that host will be marked down, and that host's address will not appear in Consul responses for that service.

Consul also automatically tracks telemetry data (runtime metrics) for various internal services. These are available from every running Agent by sending a *SIGUSR1* to the consul service and can optionally be exported to a listening StatsD daemon.

Push, Pull, Both, or Neither?

In the context of monitoring, Consul is probably best described as a push-based system. Every host in your infrastructure runs the Consul agent. Consul servers are agents running in server mode, and clients are agents running in client mode. Individual clients run health checks on themselves at the direction of the Consul server, and individual clients also gossip with neighboring nodes in their cluster group to maintain consensus on cluster health. Clients interact with the server on regular intervals or as necessary to report that a peer node has become unavailable.

Measurement Resolution

Individual node and service health check ticks are configurable and normally run on 10–30 second intervals. Consul telemetry data is aggregated at 10-second intervals and stored in memory for 1 minute.

Data Storage

Cluster-state data and other metadata is stored locally on each node in the *data-dir*. Various formats are employed. Telemetry data is retained in-memory for one minute. Consul also ships with its own KV store, which is intended to retain configuration and leader-election data but it isn't relevant in a monitoring context.

Analysis Capabilities

Consul is not a monitoring system, so traditional, monitoring-centric analysis capabilities are nascent at best. There is a web UI that depicts red/yellow/green availability data, and every running agent has a */health* API endpoint from which the current health-state can be obtained.

Notification Capabilities

As just stated, Consul is not a monitoring system, and therefore has no built-in notification support, but there are several options for implementing alerting. Atlas, which is Hashicorp's commercial product, can provide turnkey alerts from registered Consul-enabled datacenters. The built-in watchers feature can monitor reconfigured views and export events to external handlers. And, finally, the open source tool: *consul-alerts* is a Golang-based service that registers with a locally running Consul agent and implements alerts.

Integration Capabilities

Because it is not a monitoring system, Consul integrates very well with monitoring systems in general, providing built-in telemetry hooks for StatsD as well as easy-to-scrape API endpoints for health and availability data.

Scaling Model

Consul is a distributed, horizontally-scalable application.

Elasticsearch, Logstash, and Kibana (ELK)

ELK is an acronym that describes a collection of three separate open source tools: Elasticsearch, Logstash, and Kibana. All three tools are managed by Elastic, an open source company focused on creating data-analysis tools. Elasticsearch is a No-SQL database that is based on the Lucene search engine, and Kibana is a visualization layer created for use with Elasticsearch. Logstash is a log-processing tool, and arguably the only one of these tools that can properly be called a "monitoring" tool. Despite the "log" moniker, however, it acts more like passive monitoring agent, and many shops rely wholly on ELK for their monitoring needs.

Logstash ingests log data from various sources, parsing, transforming, and normalizing the text, which is then exported into Elasticsearch, where it is stored and indexed. Administrators can then interrogate the data using Kibana to, for example, graph the number of occurrences of HTTP 404s, or parse out latency numbers and graph them.

If it sounds less like running a monitoring system and more like running three separate, mildly complex pieces of software, you'd be right. ELK has many uses besides monitoring, including business analytics and security, and it comes with a nontrivial learning curve and management overhead. It is often used by teams with dedicated telemetry staff or those who have pre-existing Elasticsearch expertise.

Push, Pull, Both, or Neither?

Logstash is obviously the primary means of getting data into the system. It is modular, supporting ruby-based plug-ins for both input and output. Input plug-ins mostly pull, but depending on your use case, polling plug-ins for systems like Kafka and even S3 are available.

Measurement Resolution

Because ELK is really a free-form data-analysis tool, there is no "tick," or standard resolution.

Data Storage

Elasticsearch is a distributed document store. It literally stores data-structures that are input and returned as serialized JSON documents. You feed preparsed, formatted data from Logstash in as JSON, and Elasticsearch automatically indexes every field in each data structure. You use "Grok," a DSL built-in to Logstash to format your log data using configuration commands called "patterns." Logstash ships with around 120 patterns by default, which are capable of parsing log data from many common sources (you can see these here at *https://github.com/logstash-plugins/logstash-patterns-core/tree/master/patterns*).

Analysis Capabilities

ELK really isn't a monitoring tool, it's a system for ad hoc data analysis. Its analysis capabilities are therefore outstanding, to the point of being complete overkill for most shops with typical data-monitoring concerns. If you are the type of engineer who wishes you could very deliberately preprocess your log data, and then perform ad hoc, interactive statistical analysis on the result, ELK is certainly the tool for you. But, if you're in search of a tool to help you monitor *X*, ELK will probably seem unhelpful and frustrating.

Notification Capabilities

No real notification capabilities are built in to the ELK stack. By way of workarounds, there is a commercial tool from Elastic called "watcher" that will query Elasticsearch. There is also an `email{}` output target in Logstash and an open source tool called ElastAlert.

Integration Capabilities

Integration options pretty much center around Logstash, which has many different plug-ins for input and output. You can see these at *https://github.com/logstash-plugins*.

Scaling Model

Elasticsearch is a horizontally scalable distributed data store. It will scale as far as your pocketbook allows.

Prometheus: Graphite Reimagined

Prometheus is a metrics-savvy, open source monitoring tool with a purpose-built time series database (TSDB), it's own agent suite, multi-dimensionality, and first-class alerting support. It works by centrally polling an HTTP port on the things you want to monitor, which means the things you want to monitor need to be running one of the various Prometheus Agents (called exporters).

Exporters collect and expose metrics for various entities (System metrics, DBs, etc.), or ingest and re-export metrics from other daemons like StatsD. All of this data is made available via HTTP, and the Prometheus server is then configured to scrape these ports every so often.

Push, Pull, Both, or Neither?

Prometheus is entirely a pull-based system, using Prometheus-specific "exporters" to collect metrics on the client side and expose them via HTTP. There is an optional push gateway that you can use to inject metrics, but this works by accepting metrics via HTTP POST and re-exposing them for the Prometheus server to scrape via HTTP.

Measurement Resolution

By default Prometheus operates on a 15-second tick. This is configurable on a per/host per/metric basis.

Data Storage

Prometheus has a sophisticated custom-built local storage system which relies on LevelDB for tag indexing. Sample data is organized in chunks of constant size (1,024-byte payload). These chunks are then saved on disk in one file per time series.

As mentioned earlier in the introduction, Prometheus is one of the newer class of TSDBs that support multidimensionality. Metrics are individually identified as a combination of a name and a collection of key/value tags, which enables fast searching and queries via the web UI.

Analysis Capabilities

Support for multidimensional data tagging and querying combined with an expressive DSL that enables users to quickly compute aggregations, summaries, and percentiles make for an excellent time-series analysis system. The built-in expression browser can execute ad hoc queries and visualize individual metrics, and you can use Grafana or PromDash to create longer-lived dashboards.

Notification Capabilities

Prometheus has first-class alerting support via alerting rules that you define in the Prometheus server. You can configure multiple Prometheus servers to point to a separately running Alertmanager daemon, which handles alert processing (silencing/aggregating duplicate alerts, and so on). The usual notification targets are supported (email, SMS, PagerDuty, and so forth).

Integration Capabilities

Prometheus directly supports a broad array of language-bindings, daemons, and DBs. Third-party exporters exist for a wider range of targets like HAProxy, JMX, Redis, and so on. If you can't find the exporter you're looking for, Prometheus also has first-class support for StatsD, which can ingest pretty much anything.

Scaling Model

Prometheus' custom data store is a local-only single-point-of-failure, and its poll-only operation bring into question the system's overall scalability and resilience. The documents report single-instances with SSDs ingesting on the order of 500,000 metrics per second, and the Prometheus blog mentions million-metric-per-second ingestion.

Further, there are federation and sharding strategies that you can employ that isolate the polling burden from the processing burden. For more information, go to *http://www.robustperception.io/scaling-and-federating-prometheus*.

CHAPTER 6

traditional.free_open.monoliths.network

Not many network-centric tools appeared in the Katar survey, but SmokePing remains a popular monolithic monitoring tool with a network focus.

SmokePing: Ping, with Graphs

SmokePing is a Perl script designed to ping your network components and visualize their latency and packet loss. It is simple in both concept and execution and takes a novel approach to visualization by taking multiple samples per measurement interval (10 by default). The samples are then sorted by latency and the median latency is plotted as the actual value, whereas the actual samples are plotted in gray as deviations. The shade of gray is selected as a function of the distance from the median, and this gives the resulting charts a "smokey" appearance (hence, the tool's moniker) with darker shades of gray close to the median value, and lighter shades as the deviance increases. The median value itself is also colorized as a function of packet loss.

Push, Pull, Both, or Neither?

A single-instance SmokePing system is wholly pull-based; however, when multiple systems are employed, the "slave" nodes do their own polling and push the result to the "master" system.

Measurement Resolution

The default tick (referred to as a “step” in the documentation) is 60 seconds.

Data Storage

SmokePing was written by the author of RRDTool and uses RRDTool for storage and visualization.

Analysis Capabilities

SmokePing line-graphs do a very nice job of depicting the latency of individual network hops as well as the sigma latency of individual nodes.

Notification Capabilities

SmokePing has excellent alert definition syntax. You can, for example, craft alerts that fire when the RTT is greater than *X* for *Y* sampling periods, or even based on historic average median latency over several measurement rounds. The notifications themselves, however, are restricted to email targets only.

Integration Capabilities

You can extend SmokePing with plug-ins that add new protocols of service checks (curl instead of ping, for example) or those that fire alerts based on new types of criteria (exponential weighted average of packet loss for example).

Otherwise, the system has a few hooks for other monitoring systems to interact with it. One option is the RRDTool files themselves, and another is the included “smokeinfo” script, which provides a handy command-line-based wrapper to the RRDTool data.

Scaling Model

SmokePing masters can orchestrate many distributed nodes in order to take samples from different perspectives on the network, but there is no high availability configuration for the master node itself, so the Master is a single-point-of-failure. If it dies, all monitoring stops.

Further, the RRDTool-based storage can’t be distributed off a single host, so SmokePing is I/O bound, and scales as far as you can scale RRDTool-based storage on a single node.

CHAPTER 7
traditional.free_open.monoliths.system

By far the most popular category, the tools in this chapter are what most people envision when they think of monitoring tools: a single-process daemon that collects availability data about servers.

Check_MK: Making Nagios Easier

Check_MK began life as an easier-to-use remote execution framework for Nagios (read: agent), but it is usable today as a stand-alone tool. It would be a straightforward agent were it not for the various complications inherent in integrating with a pre-existing Nagios system. These mostly include the typical configuration Nagios needs before it can accept check results for hosts and services.

To work around Nagios' configuration complexities, Check_MK embeds a fully fledged service discovery system and Nagios configuration generator, enabling you to scan for new hosts and add them to your pre-existing Nagios configuration with a few command-line interface commands. Check_MK then follows up by adding a queriable state socket for the Nagios daemon, a replacement UI, and even a web-based Nagios configuration service, making it an extremely popular one-stop Nagios simplification system.

Push, Pull, Both, or Neither?

Depending on how you configure it, Check_MK can proxy active checks (pull) or it can operate passively (push). Even if you configure it for active checking, however, Check_MK is a far more efficient poller than Nagios core is, interacting once with each remote system it monitors per polling interval rather than once per configured service check as Nagios+NRPE would.

Measurement Resolution

Check_MK inherits its polling interval from Nagios.

Data Storage

Check_MK is a pass-through system, gathering state information and providing it upstream to Nagios Core. It doesn't have a data-storage framework of its own.

Analysis Capabilities

Check_MK doesn't add any analysis capabilities to Nagios that don't already exist. It does support and generate configuration compatible with Pnp4Nagios, which is the de facto RRDTool-based graphing framework of choice.

Notification Capabilities

Check_MK ships with a notification system for Nagios, which mostly affects how individual users are notified rather than whether they are notified. See *http://mathias-kettner.com/checkmk_flexible_notifications.html* for details.

Integration Capabilities

Check_MK adds a Nagios Event Broker module called "Livestatus," which greatly enhances Nagios' already world-class integration support.

Scaling Model

Check_MK arguably increases the performance of a single-instance Nagios host by more effectively polling remote hosts. Otherwise, its scaling model remains Nagios' scaling model.

Ganglia: Large Scale, High-Resolution Metrics Collection

Ganglia is a metrics-oriented monitoring system designed for high-performance computing clusters. It is able to efficiently collect metrics from thousands of hosts at ~10-second resolutions.

It works by using Ganglia Monitoring Daemon (gmond), a C-based agent that employs a multicast gossip protocol to collect and summarize metric data from groups of proximate hosts. These summaries are then polled by one or more central collectors called Ganglia Meta Daemon (gmetad), which might or might not in turn be polled by other collectors. Ganglia is one of the few systems that uses the rollup pattern described in Chapter 2.

Push, Pull, Both, or Neither?

Individual gmond systems in the same cluster multicast their metrics to one another on a set interval (push). Gmetad systems then collect the cluster-wide metrics summary from a single host in each cluster (pull). This arrangement minimizes the polling of individual hosts, and scales quite well.

Measurement Resolution

Ganglia runs on a 15-second tick by default.

Data Storage

Each node in the cluster keeps an in-memory copy of the entire cluster state. Metrics themselves are saved in RRDTool on the topmost gmetad server.

Analysis Capabilities

Gweb, Ganglia's web interface, is, in my opinion, the very best RRDTool web frontend. It's probably the best achievable analysis system for data that resides in RRDTool databases.

Notification Capabilities

Ganglia has no built-in alerting functionality, but gmetad does run a queryable REST API, and the project ships with a Nagios plug-in that makes it fairly trivial to use Nagios as an alerting system for Ganglia (check the contrib directory).

Integration Capabilities

Ganglia is a fairly tightly-integrated system, which was, for the most part, designed to be used with high-performance computing clusters (the configuration requires that you organize systems into clusters, for example). Gmetad has built-in support for exploring metrics to a listening Carbon daemon (Graphite), and as with any RRDTool-based tool, you can interact with the RRDs directly.

Scaling Model

Ganglia was built to scale by using the compute resources of the monitored hosts to bear most of the brunt of the collection and aggregation workload. The system is capable of handling millions of metrics from tens of thousands of hosts on a per-second scale. It does a fantastic job of remaining resilient to failure to individual nodes and gmetad is effectively stateless, so you can run multiple parallel instances without incurring any measurable load on the monitored hosts. You can also stack gmetads to further scale the rollup burden, but given the RRDTool storage, the system is ultimately not horizontally scalable and you will eventually be limited by either the size of the filesystem that holds the RRDs or the I/O capabilities of the disk that holds the RRDs.

Icinga: Nagios Extended

Icinga is a centralized-poller that was forked from the Nagios core in 2007. Like Nagios, it executes standalone monitoring scripts on a rotating schedule and it remains plug-in compatible with Nagios.

Currently there are two major version branches, Icinga v1 and v2, the primary difference being that Icinga1 still uses Nagios configuration syntax, whereas Icinga2 uses a new, more programmatic configuration syntax that is capable of tracking variables against base-objects and then assigning services/notifications/etc. based on those variables rather than statically assigning them like Nagios does.

Push, Pull, Both, or Neither?

In its default mode, Icinga, like Nagios is a pull-based system, but you can configure it to accept passive check results, as well, which implement push-based status updates (usually via HTTPS).

Measurement Resolution

Icinga inherits its resolution form Nagios, which was designed to operate on the order of minutes. By default it launches active service checks every five minutes.

Data Storage

One primary reason for the fork was Nagios' steadfast resistance to replacing the state file with a relational database. Predictably, Icinga made this change pretty much immediately after the fork, providing "IDO" modules that can store Icinga state data in either a MySQL or Postgre database. The system does not have a native means of storing and presenting performance data, but the various Nagios Perfdata add-ons all remain compatible with Icinga.

Analysis Capabilities

The default Icinga UI supports basic real-time red/yellow/green style availability data with limited historical analysis capabilities. Third-party and commercial UI add-ons exist that enable some performance data in the form of line graphs.

Notification Capabilities

By default Nagios supports email notifications, UI-based alert acknowledgments, and highly configurable escalations. It is moderately easy but not trivial to define alternate notification protocols, and third-party add-ons exist to extend it to support services like PagerDuty and VictorOps.

Integration Capabilities

Icinga is generally identical to Nagios with respect to the set of hooks typically employed by system administrators to extend Nagios to do things like implement new notification types. It is missing some post-Nagios-Core v4 hooks like the Nagios Event Radio Dispatcher interface, but the event broker interface remains intact.

It also includes several powerful hooks, made possible by the use of a database state-store that Nagios lacks. These include the Query functionality of the database itself as well a fully functional web API.

Scaling Model

Icinga scales on par with Nagios into the range the tens of thousands of active service checks on modern hardware, depending on the configured polling interval (functionally, it's not much different in this regard). It does, however, add native clustering support to achieve high-availability setups.

Monit: Think Monitoringd

Monit began as a process-monitoring system—the sort of thing you might use DAEMON Tools or, more recently, systemd to achieve. It has since grown into a more general-purpose standalone agent, that is capable of monitoring local processes; changes to files and directories; system resources like CPU, RAM, and network cards; and it can even remotely check listening services and run ad hoc scripts.

Its "killer feature" is the ability to define local automation that takes place when Monit detects a bad result from something it's monitoring. This was originally intended to restart a crashed process, but any ad hoc scripting can be executed.

Push, Pull, Both, or Neither?

Neither. There is no "server." Monit runs as a state-independent daemon process on every host in your infrastructure. All collection, notifications, and automation is run locally on each system individually.

Measurement Resolution

The default polling interval is 120 seconds (2 minutes).

Data Storage

None to speak of. As previously mentioned, Monit runs as a daemon process; it wakes up on a given interval, checks the list of entities for which it has been configured to check, and goes back to sleep. It maintains a state file on the local filesystem to retain state between reboots and enable some alerting functionality (e.g., only alert if *X* is down for two polling cycles).

Analysis Capabilities

None to speak of. Monit was designed to either alert an operator to an error state or to take immediate automated corrective action when an error state is detected. The Monit daemon runs a local HTTP server that will output system state in XML, and there are a few open source scripts out there that will scrape this data and summarize it or forward it to systems like Ganglia or Librato (see *https://github.com/karmi/monittr*). There is also a commercial product called M/Monit with a web UI.

Notification Capabilities

Monit can call arbitrary external scripts or send email when error states are detected. It lacks some of the features of other monitoring systems in this regard. The ability to acknowledge alerts at runtime is one example. Further, the specification language for what events to alert on can get a little unwieldy in edge cases.

Integration Capabilities

The Monit daemon runs a local HTTP server that will output system state in XML. There are a few open source scripts out there that will scrape this data and forward it to systems like Ganglia or Librato.

Scaling Model

Strictly speaking Monit is a stateless daemon, it is horizontally scalable in the sense that there is nothing really to scale.

Munin: Cacti for Servers

Munin is a metrics-centric centralized poller written in Perl. It uses an agent (called munin-node) to collect statistics on every system you want to monitor. Those are periodically scraped by the Munin server and saved in RRDs. Munin-node collects metrics by running one or more plug-ins. The plug-ins that ship with Munin as well as those available from their GitHub repo are almost entirely written in shell and follow simple design criteria described at *http://guide.munin-monitoring.org/en/latest/plugin/writing.html*.

Push, Pull, Both, or Neither?

Munin is a centralized poller with a dedicated agent of its own.

Measurement Resolution

The default polling interval is five minutes, and Munin has for years offered no way to change this hardcoded default. As of the 2.0 branch, the as-of-yet undocumented *update_rate* parameter allows different measurement resolutions. Your mileage may vary.

Data Storage

Munin stores all data locally to RRDs.

Analysis Capabilities

Capabilities are pretty much what you'd expect for an RRDTool-based system. You get statically configured line graphs of statically configured hosts and services.

Notification Capabilities

Although Munin has nascent notification support of its own, the preferred mechanism is to alert via Nagios using Munin's built-in support for emitting Nagios-compatible passive checks to a running NRPE daemon.

Integration Capabilities

Munin is, for the most part, a monolithic replacement for Cacti with better support for general-purpose computational entities (as opposed to Cacti's SNMP focus). It does have excellent built-in support for emitting Nagios passive-check results.

Scaling Model

Munin server is a single-node system with no built-in support for failover. It is primarily I/O and storage bound due to its dependence on RRDTool.

Naemon: The New Nagios

Naemon is another Nagios fork, born in 2013. It executes standalone monitoring scripts on a rotating schedule, just like Nagios, and is compatible with Nagios plug-ins, add-ons, and configuration syntax.

Push, Pull, Both, or Neither?

In its default mode, Naemon is a pull-based system, but you can configure it to accept passive check results, as well, which implement push-based status updates via HTTPS.

Measurement Resolution

Like Nagios, Naemon is designed to operate on the order of minutes. By default it launches active service checks every five minutes.

Data Storage

Naemon inherits Nagios' statefile storage, but adds a built-in Livestatus plug-in to provide a query API to its own in-memory state.

Analysis Capabilities

Naemon has replaced the Nagios core CGIs with Livestatus and Thruk. Analysis capabilities remain about the same. For more information, see *http://www.thruk.org*.

Notification Capabilities

Naemon inherits notification support unchanged from Nagios. It supports email notifications, UI-based alert acknowledgments, and highly configurable escalation. It is moderately easy but not trivial to define alternate notification protocols, and third party add-ons exist to extend it to support services like PagerDuty and VictorOps.

Integration Capabilities

Naemon ships with Livestatus built in; otherwise, it is unchanged from Nagios in terms of integration capablities.

Scaling Model

Naemon was forked by the team that implemented lightweight worker processes in Nagios core 4, and that team has continued its parallelization work in Naemon. It's too soon to determine how much Naemon improves over Nagios' scaling story but Naemon is generally expected to surpass Nagios in this regard.

Nagios: The Venerable, Ubiquitous, Operations-Centric, System Monitoring Monolith

Nagios is one of the oldest open source monitoring tools. It is a centralized-polling system that executes standalone monitoring scripts on a rotating schedule. Its somewhat unwieldy text-based configuration makes it highly flexible, albeit difficult, to configure, and thousands of third-party add-ons exist to extend its functionality and simplify its configuration. Nagios is generally believed to be the most widely used open source monitoring tool in the world today.

Push, Pull, Both, or Neither?

In its default mode, Nagios is a pull-based system, but it can be configured to accept passive check results, as well, which implement push-based status updates via HTTPS.

Measurement Resolution

Nagios is designed to operate on the order of minutes. By default, it launches active service checks every five minutes.

Data Storage

Nagios stores state change events only, logging whenever a service changes from one state (like OK) to another state (like warning). These are written to a log file located on the local filesystem. There is nascent built-in support for collecting performance data, and there are third-party add-ons which emit this performance data to external processors like Graphite (*/Part2/traditional/free_open/processors/data/graphite.md*) and Librato (*/Part2/hosted/freemium/*

processors/data/librato.md). Other third-party add-ons exist to replace the state log file with MySQL and PostgreSQL databases.

Analysis Capabilities

The default Nagios UI supports basic real-time red/yellow/green style availability data with limited historical analysis capabilities. Third-party and commercial UIs exist that enable some performance data in the form of line graphs.

Notification Capabilities

By default Nagios supports email notifications, UI-based alert acknowledgments, and highly configurable escalations. It is moderately easy but not trivial to define alternate notification protocols, and third-party add-ons exist to extend it to support services like PagerDuty and VictorOps.

Integration Capabilities

In some contexts, Nagios was designed with excellent "hooks" to support end-user extensions and add-ons. It is easy, for example, to create new service checks and redefine notification commands. In other context, Nagios is quite difficult to extend; for example, it is not easy to export performance data from Nagios into telemetry analysis systems like Graphite (*/Part2/traditional/free_open/processors/data/graphite.md*). Tools exist to accomplish this, but the configuration will take a first-time user several hours at a minimum. In still other contexts, Nagios was not designed for integration at all; for example, there is no API or other means to query the Nagios Daemon for real-time status updates on arbitrary hosts. Integrations that provide this functionality exist but are not trivial to install. You must write DIY solutions in C in order to communicate with the Nagios internal event broker interface.

Scaling Model

Nagios scales well into the tens of thousands of active service checks on modern hardware depending on the configured polling interval. With passive checks, it scales into the range of half a million service checks depending on the configured polling interval. Beyond that, knowledgeable, dedicated telemetry teams can design and maintain multidaemon setups by using third-party add-ons.

OMD: Nagios Made Easy(er)

OMD is a repackaging of the Nagios core along with several add-ons and replacement UIs. The packaging includes Nagios core, a fully-preconfigured Check_MK installation to accomplish remote-execution of Nagios plug-ins on remote hosts, Thruk (a popular replacement UI), MK-Multisite (IMO the best Nagios user-interface in existence), WATO (a web-based Nagios configuration tool) and more. If you are familiar with the Nagios add-on universe, you'll likely find most, if not all of your favorite Nagios add-ons already preconfigured for you in OMD.

It is a large install, but it's arguably preferable to wiring up the tool-chain manually. I say arguably because OMD itself somewhat ironically presents a few management challenges that don't exist if you decide to just roll the tool-chain yourself. You are constrained to the versions of the tools in it's current package, for example, and if you use WATO you'll need to manage the WATO configuration against whatever manual configuration you or other administrators make, which can quickly result in a split-brained, extra layer of confusion atop an already confusing Nagios core configuration.

Push, Pull, Both, or Neither?

In its default mode, Nagios is a pull-based system, but you can configure it to accept "passive check results," as well, which implement push-based status updates via HTTPS.

Measurement Resolution

Nagios is designed to operate on the order of minutes. By default it launches active service checks every five minutes. OMD adds a preconfigured PNP4Nagios implementation, which stores Nagios performance data in RRDTool databases locally on the filesystem.

By default you'll get the following data retention:

- 2,880 entries with 1-minute step = 48 hours of 1-minute resolution data
- 2,880 entries with 5-minute step = 10 days of 5-minute resolution data
- 4,320 entries with 30-minute step = 90 days of 30-minute resolution data

- 5,840 entries with 360-minute step = 4 years of 6-hour resolution data

PNP4Nagios will store three summarizations per roll-up, one averaged, one minimum and one maximum.

Data Storage

Nagios stores state change events only, logging whenever a service changes from one state (like *OK*) to another state (like *warning*). These are written to a log file located on the local filesystem. OMD adds a preconfigured PNP4Nagios implementation, which stores Nagios performance data in RRDTool databases locally on the filesystem. By default, the data storage requirement is ~1.2 MB per metric.

Analysis Capabilities

The default Nagios UI supports basic real-time red/yellow/green–style availability data with limited historical analysis capabilities. OMD adds a preconfigured PNP4Nagios implementation in addition to integrations with Thruk, MK-Multisite, and the core UI. This adds real-time RRDTool graphs in the Core UI, and per-service "Fuel-gauge" visualizations for Thruk and MK-Multisite.

Notification Capabilities

By default Nagios supports email notifications, UI-based alert acknowledgments, and highly configurable escalations. It is moderately easy but not trivial to define alternate notification protocols, and third-party add-ons exist to extend it to support services like PagerDuty and VictorOps.

Integration Capabilities

In some contexts, Nagios was designed with excellent "hooks" to support end-user extensions and add ons. It is easy, for example, to create new service checks and redefine notification commands. In other contexts, Nagios is quite difficult to extend; for example, it is not easy to export performance data from Nagios into telemetry analysis systems like Graphite (*/Part2/traditional/free_open/processors/data/graphite.md*). Tools exist to accomplish this, but the configuration will take a first-time user several hours at a minimum. In still other contexts, Nagios was not designed for integration at all; for example, there is no API or other means to query the Nagios Daemon for real-time status updates on arbitrary hosts.

OMD makes up for many of these deficiencies. It includes MK-Livestatus, for example, which provides real-time Nagios state data via a queryable network socket. Its data exportation capabilities are limited to RRDTool, however.

Scaling Model

Nagios scales well into the tens of thousands of active service checks on modern hardware, depending on the configured polling interval. With passive checks, it scales into the range of half a million service checks, depending on the configured polling interval.

OMD uses the Check_MK agent for remote checks, which uses passive checks by default. It also includes a Mod-Gearman installation, which advanced users can employ to build distributed and high-availability Nagios infrastructures.

Sensu: Nagios Reimagined

Sensu is a centralized poller that brings modern distributed-systems engineering tools and techniques to bear in order to overcome some of the limitations inherent in the centralized-poller design. The foremost of these is probably scalability. Sensu introduces a pub/sub system and centralized message queue to remove single points of failure and distribute the monitoring load across many systems.

Sensu-server acts as the centralized orchestration point, keeping track of the schedule and publishing service checks as job-feeds, to which the Sensu-clients subscribe. Client systems push their check results on to a queue in the form of events, which are processed by systems configured as Sensu event handlers.

The system uses JSON configuration syntax and IPC, interacts very well with configuration management engines, and although it’s Nagios-plug-in compatible, includes a built-in means of easily installing ad hoc check plug-ins from its own extensive collection of plug-ins from the command line (via bundler). There are a lot of moving parts, but it is a wonderful system to work with when you’ve wrapped your head around it.

Push, Pull, Both, or Neither?

Sensu is best described as a hybrid push/pull system. It's worth noting that the centralized queue keeps all of the CPU-intensive work distributed, so even when the system is polling, it's polling the queue rather than individual client systems individually.

Measurement Resolution

Sensu was designed to collect measurements on the order of seconds (overcoming another limitation normally encountered with traditional pollers). The polling interval is set per-check and most of the boilerplate configuration uses a 10-second polling interval.

Data Storage

As mentioned, Sensu relies on a centralized queue for inter-system communication as well as a Redis KV store to house state data. You can use Redis for both of these purposes or RabbitMQ for the queue and Redis for the state store.

Analysis Capabilities

Sensu very intentionally relegates itself to the role of distributed, scalable data collector and assumes that your use of specialized, third-party tools for data analysis. This is, in my opinion, a wise, and very welcome decision.

Notification Capabilities

Alerts are implemented as events. They're passed to handlers process to process, and Sensu assumes that the handler knows what to do with it. Handlers come in a few types, one of which is the "pipe" handler, which simply pipes the JSON-encoded event into the program you specify (there are many community-provided handlers you can use, you don't need to write them yourself). An event will fire on any check that returns a nonzero exit code. That event will be handled either by the default event handler, or the handler(s) you specify in the check configuration.

This is, by the way, exactly the same way Sensu handles performance data export: the check fires an event containing performance data, which is picked up by the handler you specify in the check configuration.

Integration Capabilities

Sensu is also heavily invested in being a fantastic distributed data collector and therefore has fantastic integration features (it is certainly king among the centralized pollers in this regard). Apart from the wide-open event framework, it boasts myriad web-based APIs that can be used for state and service introspection.

Scaling Model

Sensu is also heavily invested in being scalable. Its sole limitation being its reliance on Redis for state data. For detailed information, see Sensu's own scaling page at *https://sensuapp.org/docs/0.16/scaling_strategies*.

Shinken: Py-Nagios

Shinken is a from-scratch rewrite of Nagios core in Python. Unlike the other Nagios forks, Shinken is committed to configuration-compatibility with Nagios core, which means your existing Nagios configuration will work without modification in Shinken, as will (obviously) your existing Nagios plug-ins.

Additionally, Shinken adds some much needed HA and scaling options to the Nagios architectures, allowing you to run schedulers and pollers across multiple systems while centrally maintaining state. It also extends Nagios' configuration syntax in interesting and useful ways (e.g. enabling you to specify *x out of y* logic for clusters inside the service check configuration). Shinken also adds numerous easy-to-install, turn-key integration support for things like Graphite, and is, in general, easier from which to extract state data.

Push, Pull, Both, or Neither?

Shinken supports both the active and passive check models with which Nagios administrators will be familiar (both).

Measurement Resolution

Shinken is based on Nagios and therefore designed to run on a tick of one minute or longer.

Data Storage

Shinken is highly configurable with respect to storing state, logs, and metrics from the entities it monitors. This support is provided as a series of easy-to-install modules. You can see a complete list at *http://shinken.io/browse/modules/updated*.

Analysis Capabilities

Shinken ships with its own standalone web UI and boasts built-in support for both RRDTool and Graphite, which is pretty great.

Notification Capabilities

Shinken pretty much identically reimplements Nagios' notification framework, to include all of the macros and command syntax.

Integration Capabilities

Shinken has all of the integration potential of Nagios and much more. You can, with the one-liner installation of some combination of modules, export data to pretty much any system with which you might want to communicate. In addition, the native Livestatus support provides a queryable state socket that, in my opinion, should have been a standard feature in Nagios core 4.

Scaling Model

Shinken is desinged internally to be much more modular than Nagios, and although some of these components can't be distributed (e.g. the Arbiter and Broker daemons), the components that typically represent scaling challenges to Nagios' design can be horizontally scaled. For detailed scaling information, go to *http://shinken.readthedocs.io/en/latest/09_architecture/the-shinken-architecture.html*.

Xymon: Bigger Big Brother

Xymon grew out of the bbgen toolkit for Big Brother. It is a higher-performance clone of Big Brother, and aside from some performance-based changes like centralized configuration, it looks and feels very similar. Xymon is a classic centralized poller, written in C. It is composed of the Unix-only agent called Xymon Client, and the Xymon server process. Aside from its own agent, the server also has native support for monitoring network sockets and SNMP.

Push, Pull, Both, or Neither?

Xymon is pull based.

Measurement Resolution

The default polling interval is five minutes.

Data Storage

Xymon stores metrics data in RRDTool locally on the server.

Analysis Capabilities

Analysis capabilities are nascent. Apart from the dozen or so prefab reports, the web UI is rigidly hierarchal. You click down into individual hosts until you're given RRDTool graphs that depict individual services on those hosts.

Notification Capabilities

Xymon provides email-based, or ad hoc script-based alerting on simple thresholds on individual service or host problems. Acknowledgments are supported, but time-period squelching is not.

Integration Capabilities

Xymon is a strictly monolithic system. You can interact directly with the RRDs it writes, but otherwise there are no integration hooks.

Scaling Model

Xymon server is a single host with RRDTool storage. There is no HA configuration.

Zabbix: A Nagios Replacement for "Enterprise" Businesses

Zabbix is a centralized poller designed with the needs of enterprise business customers. It is a well-documented system that configures via web forms and features first-class support services. It comprises the Zabbix server process, which schedules checks, reaps results and emits notifications; 0 or more Zabbix Proxies which distribute the check-processing load; a database instance, and the web UI.

You can't run these parts on separate systems or on the same system, which makes Zabbix a bit more scalable than its direct competitors but also arguably a little more difficult to install and reason about.

Push, Pull, Both, or Neither?

Zabbix agent's run on every system you want to monitor. The agents can either actively run their own checks on the monitored systems or perform one-off checks at the direction of the Zabbix server. In either case, the system is predominantly pull-based, though it scales slightly better than other pollers like Nagios because of it's proxy services and affinity for pushing check-logic out to the monitored hosts.

Measurement Resolution

The default polling interval in most cases is one-minute; however, Zabbix makes it easier than most centralized pollers to specify sub one-minute polling intervals.

Data Storage

All data is stored in a centralized SQL database (Microsoft SQL, MySQL and PostgreSQL are all supported). By default, Zabbix stores raw resolution data (history) for 30 days, and summarized data (trends) for 90 days.

Analysis Capabilities

Analysis capabilities in Zabbix are better than most of its centralized-poller brethren (which is not a particularly high bar). Many of the basic UI deficiencies one expects in RRDTool-based UIs exist in Zabbix (multiple-y axis, depicting several different metrics in the same graph), but the API can make up for many of these (see, for example *https://dzone.com/articles/escaping-zabbix-ui-pain-how*).

Notification Capabilities

Zabbix has very strong alert and notification criteria and supports basic repeat-notification and dependency-based message squelching. Add-on support for Pagerduty and VictorOps is available but more difficult to install than many other systems.

Integration Capabilities

Zabbix has strong integration capabilities that center around its REST API and SQL underpinnings. For a list of third-party add-ons, go to *http://www.zabbix.com/third_party_tools.php*.

Scaling Model

Zabbix relies on a single-point-of-failure SQL database backend, so it is generally DB I/O bound. Typical production systems service on the order of 4,000 to 6,000 hosts. For tips on scaling more then 10,000 nodes, go to *https://www.zabbix.com/forum/showthread.php?t=25349*.

CHAPTER 8

traditional.free_open.processors.data

The “new breed” of metrics-centric, open source monitoring systems more or less all categorize here. These tools typically are agnostic with regard to the data they recieve, and they do an excellent job of visualizing metrics.

Grafana: The “Uber” of Metric Frontends

Grafana is a savvy, modular web frontend for a host of metrics-oriented monitoring systems, including Graphite, InfluxDB, OpenTSDB and Prometheus. It ships with a backend server written in GO and uses Flot in the browser to plot the data. Compared to all of its open source competition, and even most of its commercial competition, Grafana is a far more elegant and user-friendly metrics UI, enabling you to explore, find, and visualize ad hoc metrics from many different backend monitoring systems, quickly and effectively.

Push, Pull, Both, or Neither?

Not applicable. Grafana queries already collected data at rest in a time series database.

Measurement Resolution

Measurement resolution depends on the underlying data store you’re using as well as the data itself. Generally speaking Grafana will plot whatever you are able to measure, but it does have features like *MaxDataPoints* to protect you from accidently making queries that result in an overabundance of browser-choking data.

Data Storage

Grafana can store metadata (dashboard configurations, user credentials, etc.) in an embedded sqlite3 database, MySQL, or Postgres. The primary underlying data store for your metrics is obviously up to you.

Analysis Capabilities

Analysis is literally Grafana's one job, and it does it extremely well. It uses backend-specific query interfaces, most of which support autocompletion to enable you to quickly and easily query metrics from your backend data stores based on tags, names, or whatever the backend supports. It can plot any combination of data sources across multiple backend metrics databases, and it comes with a plug-in architecture to enable easy visualization extensions (yes, you can have pie-charts if you'd like). Included visualization types include lines, bars, area graphs, big-numbers, and ad hoc text.

Notification Capabilities

Alerting is currently in the process of being designed and implemented in Grafana. For more information, go to *https://github.com/grafana/grafana/issues/2209*.

Integration Capabilities

Grafana was designed from the ground up to integrate with other open source tools. It is extremely modular internally and includes an API and command-line tool.

Scaling Model

Another somewhat non applicable category in the context of Grafana. Data-collection and persistence problems are really what effect scale. There is no built-in high availability functionality.

Graphite: Everybody's Favorite OSS Metrics Tool

Graphite is a metrics storage and display system. It is conceptually similar to RRDTool, storing metrics in ring-buffer databases locally on the filesystem. However, Graphite makes some critically important design leaps by doing the following:

- Accepting metrics via a trivial text-based protocol over a network socket
- Automatically configuring and creating new ad hoc metrics with sane defaults

This allows operators to isolate the metrics processing burden from the rest of the monitoring systems and enables anyone or anything that can speak the wire protocol to create and work with new metrics with no configuration overhead. Graphite has, as a result, become the most widely adopted metrics-processing system today.

Push, Pull, Both, or Neither?

Graphite listens on a network socket with *carbon*, its network listener daemon, or carbon-relay, its HA sharding counterpart. The system is entirely push-based, laying passively in wait for other systems to push metrics to it.

Measurement Resolution

Graphite was designed to run with one second or greater resolution metrics. Roll-ups and summarizations are user defined and performed in the persistence layer by *Whisper*, Graphite's custom-built TSDB.

Data Storage

As was just mentioned, Graphite uses Whisper, a simple ring-buffer metric-per-file TSDB that was purpose-created for Graphite. It is conceptually similar to RRDTool's RRDs, but implemented entirely in Python, and it comes with a far more flexible configuration design. You can, for example, set global default roll-up values that are overridden by regex-matched metric names. Graphite's data storage tier is modular, and a few other DBs (Ceres, Cassandra via Cyanite, and KairosDB) are also supported.

Analysis Capabilities

Graphite (even without the myriad frontends that augment it's analysis capabilities) is an excellent choice for metrics aggregation and analysis. The system was designed from the ground up to mix and match data from ad hoc sources into the same chart. It supports split and logarithmic axis and ships with a huge number of data transformation plug-ins that enable you to, for example, compare a signal to itself week-over-week or display the top 10 of 100 given signals, and so on.

Notification Capabilities

None, the best option is probably Bosun.

Integration Capabilities

Graphite is so ubiquitous that even most of its direct competitors have integration support for it. Many frontends and integrations exist that take graphite data and embed it. Gweb's API is excellent, and obviously the system can injest metrics from anything.

Scaling Model

Whisper DBs are a local-filesystem storage technology and this is the main impediment to scaling Graphite. You can achieve HA, as well as something akin to horizontal scaling, however, through the use of carbon-relay and some common web-scaling tools like haproxy and memcached. Federated Graphite installs do run in the wild, however you'll probably need dedicated telemetry staff to manage them. For more information, go to *https://gist.github.com/obfuscurity/63399584ea4d95f921e4*.

OpenTSDB: Hadoop All the Metrics

OpenTSDB is the brute force answer to the "Big Data" problem of metrics processing. If you've ever been frustrated by the data aggregation and roll-up problems I spoke briefly about in Chapter 1 (and you have an unlimited amount of computing resources at your disposal) You'll be happy to hear that OpenTSDB does no data summarization whatsoever. It ingests millions of millisecond precision metrics and stores them as RAW data points. You never lose precision and make none of the compromises that are usually inherent to TSDBs.

The bad news is that OpenTSDB achieves this by relying on Hadoop and Hbase to map-reduce the metrics processing and query load.

Yep, you read that correctly; OpenTSDB is literally a distributed Map-reduce infrastructure for ingesting, processing, and retrieving metrics data. After it's installed, it listens on a network socket and uses a simple text-based protocol for metrics submission. It also supports arbitrary *tagging* of metrics with key/value pairs to make them easier to look up later.

Push, Pull, Both, or Neither?

OpenTSDB is mostly a push-based system for metrics ingestion, though things get complicated quickly as you begin to distribute it across hosts and datacenters. The complications, however, are related to data replication rather than data collection or polling. OpenTSDB is just a TSDB, it doesn't come with an agent, and it does not measure anything directly.

Measurement Resolution

Millisecond precision is possible but not recommended.

Data Storage

As mentioned, the primary data store is HBase by default, although Cassandra and BigTable are also options. The documentation claims that an individual measurement takes 12 bytes on disk (with LZO compression enabled) making 100-plus billion data points per terabyte possible. Tags are stored in-line (not in external indexes) so adding tags increases the primary data storage burden.

Analysis Capabilities

OpenTSDB requires more than the average degree of expertise on the part of its users (see *http://opentsdb.net/docs/build/html/user_guide/query/index.html*). It's built-in web UI is also notoriously disliked, but Grafana is an officially supported replacement UI. Given a good frontend and a savvy end user its data-analysis capabilities are excellent.

Notification Capabilities

None. The best option is probably Bosun, but Nagios is also an officially supported option.

Integration Capabilities

Many systems include native support for OpenTSDBs wire protocol, and there are a few web UIs (you probably want Grafana).

Scaling Model

Built atop literal map-reduce infrastructure, OpenTSDBs scaling model is unparalleled but far from trivial to implement.

CHAPTER 9

traditional.free_open.processors.network

The traditional open source, network-centric data processors precursor their data-centric cousins. Because SNMP was so widely adopted by network hardware vendors, metrics collection was an obvious and effective way to understand them.

Cacti: Bringing Joy to NetOps Since 1996

Cacti is one of the first metrics-centric monolithic monitoring tools. It's a centralized poller built within a PHP app with old-school, static, web-form-based configuration. Cacti has always been a bit inflexible and unwieldy for systems administrators, but its first-class support for SNMP and RRDTool continues to make it extremely popular with the network-operations crowd to this day.

Push, Pull, Both, or Neither?

Cacti is a centralized poller. It polls via Cron using an included PHP script.

Measurement Resolution

Being a Cron-based poller, Cacti is capable only of intervals greater than a minute. The default polling interval is five minutes.

Data Storage

Cacti uses a MySQL database to house meta-data and RRDTool to store metrics.

Analysis Capabilities

Cacti's UI is based on RRDTool Graphs. It doesn't make the mistake of making you dig to find graphs, and the UI is comparatively useful (versus systems like Xymon or MRTG), but it suffers from the normal litany of RRDTool problems, including required preconfiguration, no means of ad hoc adjusting axis, no ad hoc data transformation support, and no easy means of plotting multiple signals on the same chart.

Notification Capabilities

Cacti has nascent support for sending email alerts on static thresholds via an external SNMP server.

Integration Capabilities

Cacti is a strictly monolithic system. You can interact with the RRDs that it writes.

Scaling Model

Cacti is a single-instance server with two single-point-of-failure data stores.

CHAPTER 10

traditional.free_open.processors.system

Tools in this category are breaking new ground, providing a general-purpose tool-chain for a problem that is commonly faced by data-centric organizations of any scale: how do we take metrics data from all of these various sources, and combine them to create a common telemetry signal?

Riemann: The Monitoring Leatherman

Riemann is an expressive and powerful stream-processing system for monitoring data. It ingests "events," which are protobuf-encoded objects that represent state changes (OK, WARN, etc.), or metrics (foos:4, etc.). These events are then fed through a series of nested filters that can do all sorts of interesting things with them like enumarating them, joining them together, sending emails based on their content, forwarding them to visualization systems, and so on. (The sky is the limit.)

Many different clients and language bindings exist to help you transform whatever ad hoc monitoring data you have into Riemann events and emit them into Riemann. The configuration file literally is a Clojure program, so some familiarity with Clojure is recommended; however the documentation includes a primer that will have anyone who can program in any language up and running fairly quickly.

Riemann is a difficult piece of software to blithely sum up. It is conceptually simple, and yet basically impossible for a non-programming systems administrator to comprehend and use. I use it all the time and highly recommend it.

Push, Pull, Both, or Neither?

Riemann is strictly a push-based system.

Measurement Resolution

Riemann event struts measure time in EPOC seconds, so although the system does not operate on a tick, per se, it can't distinguish between two otherwise identical events that occurred milliseconds apart (within the same EPOC second).

Data Storage

Riemann maintains an in-memory state index (internally, a non-blockinghashmap) which is queryable via the ingestion interfaces. This forms the basis of several different Riemann UIs.

Analysis Capabilities

Riemann isn't an analysis system, as such, but presents a better basis for data analysis than most monolithic monitoring tools, commercial or open source. That said, it also presents a higher learning curve than pretty much any other monitoring tool.

Notification Capabilities

Being a programmatic system by design, notification capabilities are basically limitless.

Integration Capabilities

Riemann was created explicitly to wire monitoring tools to other monitoring tools; it's integration support is unparalled.

Scaling Model

It’s difficult to talk about Riemann consistency. The “too long; didn’t read” is that the Riemann protocol lends itself well to constructing your own distributed pipeline processing (e.g., forward Riemanns to other Riemanns ad infinitum). The system is largely stateless and transient anyway, so any sort of sharding is also possible. That said, Riemann itself doesn’t provide any safety guarantees on top of what you’ve constructed, and it doesn’t provide any primitives to help you make it safe. Have fun!

CHAPTER 11

Still Reading, Eh?

Well if you've made it this far, chances are you were hoping for more content. Again, I'll invite you to check out the online version of this work at *http://github.com/librato/taxonomy*. Which reminds me, I could really use your help. If you read something in this report that you found inacurate, or if you'd like to see you favorite tool included, feel free to clone the repo and shoot me a pull request!

About the Author

Dave Josephsen is an ops engineer at Librato. He hacks on tools and infrastructure, writes about statistics, systems monitoring, alerting, metrics collection and visualization, and generally does anything he can to help other engineers close the feedback loop in their systems. He's written books for Prentice Hall and O'Reilly, speaks shell, Go, C, Python, Perl, a little bit of Spanish, and has never lost a game of Calvinball.