iVoyeur
Monitoring Design Patterns

DAVE JOSEPHSEN

Dave Josephsen is the sometime book-authoring developer evangelist at Librato.com. His continuing mission: to help engineers worldwide close the feedback loop. dave-usenix@skeptech.org

Plutarch tells a great story about Hannibal, in the time of the reign of Dictator Fabius. Hannibal, a barbarian from the south, was roaming about the countryside making trouble as barbarians do, when through either some subtlety on the part of Fabius or just seriously abominable direction-giving on the part of a few sheep farmers (reports differ), Hannibal managed to get himself trapped and surrounded in a valley.

Knowing that he wasn't going to be able to escape by assault in any direction, Hannibal came up with an ingenious ploy. The story goes that he waited until nightfall, and in the darkness he sent his men up the hillsides, while keeping his livestock in the valley floor. Then, setting the horns of his cattle ablaze, he stampeded them towards Fabius' lines. The Romans, seeing what they thought was a charge of torch-wielding barbarians (but which was really a multitude of terrified flaming cattle), reformed and dug in, and in so doing, allowed Hannibal's men to escape past them on the slopes above.

Unless you're a cow, you have to agree that this was a pretty great bit of ingenuity. It's also a pretty great example of the UNIX principle of unexpected composition: that we should craft and use tools that may be combined or used in ways that we never intended. Before Hannibal, few generals probably considered the utility of flaming cattle in the context of anti-siege technology.

The monitoring systems built in the past fail pretty miserably when it comes to the principle of unexpected composition. Every one of them is born from a core set of assumptions—assumptions that ultimately impose functional limits on what you can accomplish with the tool. This is perhaps the most important thing to understand about monitoring systems before you get started designing a monitoring infrastructure of your own: Monitoring systems become more functional as they become less featureful.

Some systems make assumptions about how you want to collect data. Some of the very first monitoring systems, for example, assumed that everyone would always monitor everything using SNMP. Other systems make assumptions about what you want to do with the data once it's collected—that you would never want to share it with other systems, or that you want to store it in thousands of teensy databases on the local file system. Most monitoring systems present this dilemma: They each solve part of the monitoring problem very well but wind up doing so in a way that saddles you with unwanted assumptions, like SNMP and thousands of teensy databases.

Many administrators interact with their monitoring infrastructures like they might a bag of jellybeans—keeping the pieces they like and discarding the rest. In the past few years, many little tools have come along that make this functionally possible, enabling you to replace or augment your single, centralized monitoring system with a bunch of tiny, purpose-driven

data collectors wired up to a source-agnostic storage tier (disclaimer: I work for a source-agnostic storage tier as a service company called Librato).

This strategy lets you use whatever combination of collectors makes sense for you. You can weave monitoring into your source code directly, or send forth herds of flaming cattle to collect it, and then store the monitoring data together, where it can be visualized, analyzed, correlated, alerted on, and even multiplexed to multiple destinations regardless of its source and method of collection.

Tons of open source data collectors and storage tiers are available, but instead of talking about any of them specifically, I'd like to write a little bit about the monitoring "patterns" that currently exist in the wild, because although there are now eleventybillion different implementation possibilities, they all combine the same basic five or six design patterns. In the same way I can describe Hannibal's livestock as a "diversion movement to break contact," I hope that categorizing these design patterns will make it easier to write about the specifics of data collectors and flaming cows later on.

Collection Patterns for External Metrics

I begin by dividing the target metrics themselves into two general categories: those that are derived from within the monitored process at runtime, and those that are gathered from outside the monitored process. Considering the latter type first, four patterns generally are used to collect availability and performance data from outside the monitored process.

The Centralized Polling Pattern

Anyone who has worked with monitoring systems for a while has used centralized pollers. They are the archetype design—the one that comes to mind first when someone utters the phrase "monitoring system" (although that is beginning to change). See Figure 1.

Like a grade-school teacher performing the morning roll call, the centralized poller is a monolithic program that is configured to periodically poll a number of remote systems, usually to ensure that they are available but also to collect performance metrics. The poller is usually implemented as a single process on a single server, and it usually attempts to make guarantees about the interval at which it polls each service.

Because this design predates the notion of configuration management engines, centralized pollers are designed to minimize the amount of configuration required on the monitored hosts. They may rely on external connectivity tests, or they may remotely execute agent software on the hosts they poll; in either case, however, their normal mode of operation is to periodically pull data directly from a population of monitored hosts.

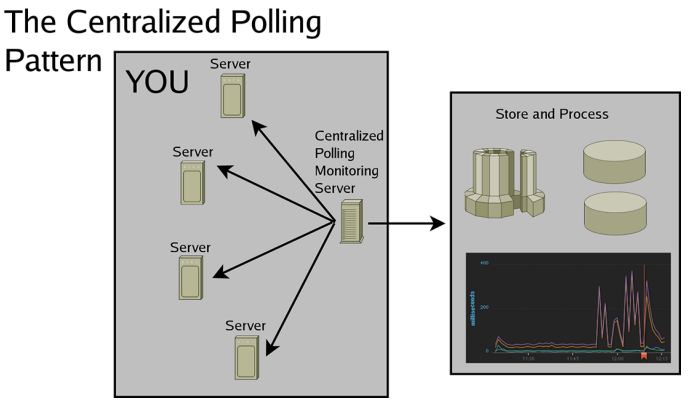

Figure 1: The centralized poller monitoring pattern

Centralized pollers are easy to implement but often difficult to scale. They typically operate on the order of minutes, using, for example, five-minute polling intervals, and this limits the resolution at which they can collect performance metrics. Older centralized pollers are likely to use agents with root-privileged shell access for scripting, communicate using insecure protocols, and have unwieldy (if any) failover options.

Although classic centralized pollers like Nagios, Munin, and Cacti are numerous, they generally don't do a great job of playing with others because they tend to make the core assumption that they are the ultimate solution to the monitoring problem at your organization. Most shops that use them in combination with other tools interject a metrics aggregator like statsd or other middleware between the polling system and other monitoring and storage systems.

The Stand-Alone Agent Pattern

Stand-alone agents have grown in popularity as configuration-management engines have become more commonplace. They are often coupled with centralized pollers or roll-up model systems to meet the needs of the environment. See Figure 2.

Agent software is installed and configured on every host that you want to monitor. Agents usually daemonize and run in the background, waking up at timed intervals to collect various performance and availability metrics. Because agents remain resident in memory and eschew the overhead of external connection setup and teardown for scheduling, they can collect metrics on the order of seconds or even microseconds. Some agents push status updates directly to external monitoring systems, and some maintain summary statistics that they present to pollers as needed via a network socket.

Agent configuration is difficult to manage without a CME, because every configuration change must be pushed to all applicable monitored hosts. Although they are generally designed

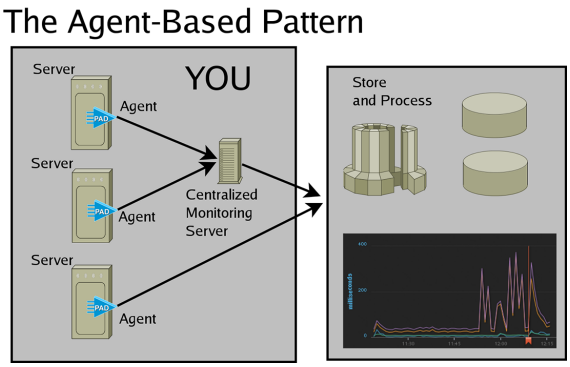

Figure 2: The stand-alone agent pattern

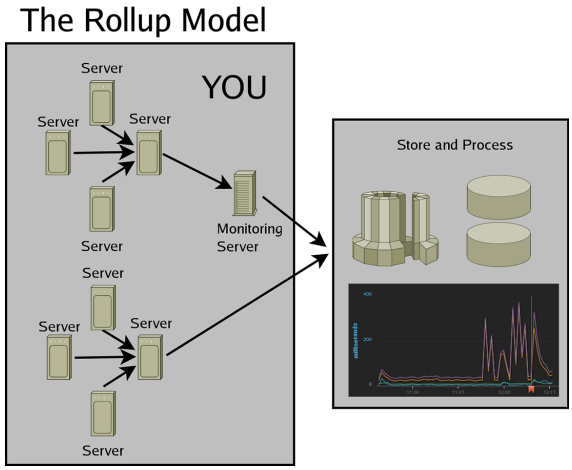

Figure 3: The roll-up pattern

to be lightweight, they can introduce a non-trivial system load if incorrectly configured. Be careful with closed-source agent software, which can introduce backdoors and stability problems. Open source agents are generally preferred because their footprint, overhead, and security can be verified and tweaked if necessary.

Collectd is probably the most popular stand-alone agent out there. Sensu uses a combination of the agent and polling pattern, interjecting a message queue between them.

The Roll-Up Pattern

The roll-up pattern is often used to achieve scale in monitoring distributed systems and large machine clusters or to aggregate common metrics across many different sources. It can be used in combination with agent software or instrumentation. See Figure 3.

The roll-up pattern is a strategy to scale the monitoring infrastructure linearly with respect to the number of monitored systems. This is usually accomplished by co-opting the monitored machines themselves to spread the monitoring workload throughout the network. Usually, small groups of machines use an election protocol to choose a proximate, regional collection host, and send all of their monitoring data to it, although sometimes the configuration is hard-coded.

The elected host summarizes and deduplicates the data, then sends it up to another host elected from a larger region of summarizers. This host in turn summarizes and deduplicates it, and so forth.

Roll-up systems scale well but can be difficult to understand and implement. Important stability and network-traffic considerations accompany the design of roll-up systems.

Ganglia is a popular monitoring project that combines stand-alone agents with the roll-up pattern to monitor massive clusters of hosts with fine-grained resolution. The statsd daemon process can be used to implement roll-up systems to hand-off in-process metrics.

Logs as Event-Streams

System and event logs provide a handy event stream from which to derive metric data. Many large shops have intricate centralized log processing infrastructure from which they feed many different types of monitoring, analytics, event correlation, and security software. If you're a Platform-as-a-Service (PaaS) customer, the log stream may be your only means to emit, collect, and inspect metric data from your application.

Applications and operating systems generate logs of important events by default. The first step in the log-stream pattern requires the installation or configuration of software on each monitored host that forwards all the logs off that host. EventReporter for Windows or rsyslogd on UNIX are popular log forwarders. Many programming languages also have log generation and forwarding libraries, such as the popular Log4J Java library. PaaS systems like Heroku have likely preconfigured the logging infrastructure for you.

Logs are generally forwarded to a central system for processing, indexing, and storage, but in larger environments they might be MapReduced or processed by other fan-out style parallel processing engines. System logs are easily multiplexed to different destinations, so there is a diverse collection of software available for processing logs for different purposes.

Although many modern syslog daemons support TCP, the syslog protocol was originally designed to use UDP in the transport layer, which can be unreliable at scale. Log data is usually emitted by the source in a timely fashion, but the intermediate processing systems can introduce some delivery latency. Log

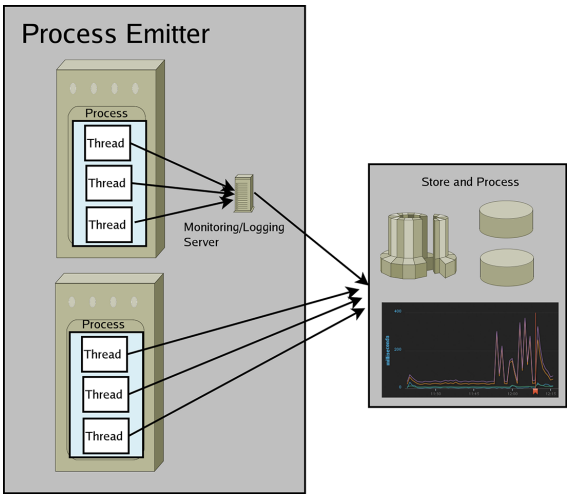

Figure 4: The process emitter pattern

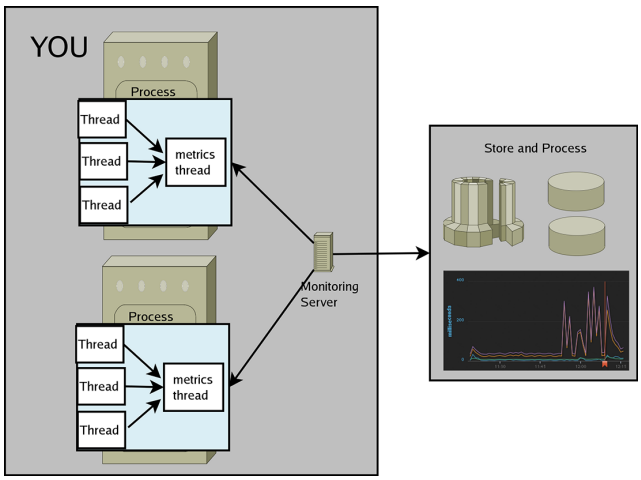

Figure 5: The process reporter pattern

data must be parsed, which can be a computationally expensive endeavor. Additional infrastructure may be required to process logging event streams as volume grows.

As I've already mentioned, with PaaS providers like Heroku and AppHarbor, logs are the only means by which to export and monitor performance data. Thus, many tools like Heroku's own log-shuttle and l2MET have grown out of that use-case. There are several popular tools for DIY enterprise log snarfing, like logstash, fluentd, and Graylog, as well as a few commercial offerings, like Splunk.

Collection Patterns for In-Process Metrics

Instrumentation libraries, which are a radical departure from the patterns discussed thus far, enable developers to embed monitoring into their applications, making them emit a constant stream of performance and availability data at runtime. This is not debugging code but a legitimate part of the program that is expected to remain resident in the application in production. Because the instrumentation resides within the process it's monitoring, it can gather statistics on things like thread count, memory buffer and cache sizes, and latency, which are difficult (in the absence of standard language support like JMX) for external processes to inspect.

Instrumentation libraries make it easy to record interesting measurements inside an application by including a wealth of instrumentation primitives like counters, gauges, and timers. Many also include complex primitives like histograms and percentiles, which facilitate a superb degree of performance visibility at runtime.

The applications in question are usually transaction-oriented; they process and queue requests from end users or external peer processes to form larger distributed systems. It is critically important for such applications to communicate their performance metrics without interrupting or otherwise introducing latency into their request cycle. Two patterns are normally employed to meet this need.

The Process Emitter Pattern

Process emitters attempt to immediately purge every metric via a non-blocking channel. See Figure 4.

The developer imports a language-specific metrics library and calls an instrumentation function like time() or increment(), as appropriate for each metric he wants to emit. The instrumentation library is effectively a process-level, stand-alone agent that takes the metric and flushes it to a non-blocking channel (usually a UDP socket or a log stream). From there, the metric is picked up by a system that employs one or more of the external-process patterns.

Statsd is a popular and widely used target for process emitters. The project maintains myriad language bindings to enable the developer to emit metrics from the application to a statsd daemon process listening on a UDP socket.

The Process Reporter Pattern

Process reporters use a non-blocking dedicated thread to store their metrics in an in-memory buffer. They either provide a concurrent interface for external processes to poll this buffer or periodically flush the buffer to upstream channels. See Figure 5.

The developer imports a language-specific metrics library and calls an instrumentation function like time() or increment(), as appropriate for each metric he wants to emit. Rather than purging the metric immediately, process reporters hand the metric off to a dedicated, non-blocking thread that stores and sometimes processes summary statistics for each metric within the

memory space of the monitored process. Process reporters can push their metrics on a timed interval to an external monitoring system or can export them on a known interface that can be polled on demand.

Process reporters are specific to the language in which they are implemented. Most popular languages have excellent metrics libraries that implement this pattern. Coda Hale Metrics for Java, Metriks for Ruby, and go-metrics are all excellent choices.

Thanks for bearing with me once again. I hope this article will help you identify the assumptions and patterns employed by the data collectors you choose to implement in your environment, or at least get you thinking about the sorts of things you can set aflame should you find yourself cornered by Roman soldiers. Be sure to check back with me in the next issue when I bend yet another tenuously related historical or mythical subject matter to my needs in my ongoing effort to document the monitoringosphere.