Logging infrastructure

Filippo Giunchedi / Keith Herron

Status: final / 2018-09-27

Introduction

As systems become more distributed and complex so does the complexity of debugging. Traditionally, applications logs are stored in plain text files on the same host the application is running on. However, in a distributed system scenario, individually checking application logs across multiple hosts is time consuming and error-prone. To address this the so-called ELK stack (**E**lasticsearch, **L**ogstash, **K**ibana) has been adopted at Wikimedia Foundation. Initially spearheaded by Bryan Davis in 2015 as a grassroot project, the ELK stack (now renamed to Elastic Stack) provides a single interface for browsing production logs, and has proven robust enough to receive continued maintenance by the Search Platform team (and its predecessor, Discovery). Starting FY 2018/2019 the SRE Infrastructure Foundations team has taken the maintainer role for the logging infrastructure.

The current architecture of logging is composed of a mixture of physical and virtual hosts to run all ELK components, Logstash is used to receive inbound logs through a variety of protocols and formats: the most common are syslog with plain and JSON payloads, GELF and "JSON lines" over UDP or TCP.

While this architecture has worked so far it has a number of shortcomings, notably most protocols involved are unreliable and plain text. Also Logstash itself in our experience hasn't been very performant in handling sudden spikes of UDP traffic, resulting in dropped packets (and thus logs). Also the current ELK deployment is present only in one datacenter (eqiad).

Goals

- Design a logging infrastructure that will support Tech at least for the next three to five years.
- Address scalability concerns and volume of logs.
- Ensure availability and reliability is greater than or equal to what we currently have.
- Define a common and simple to use interface to the logging system for applications to use.

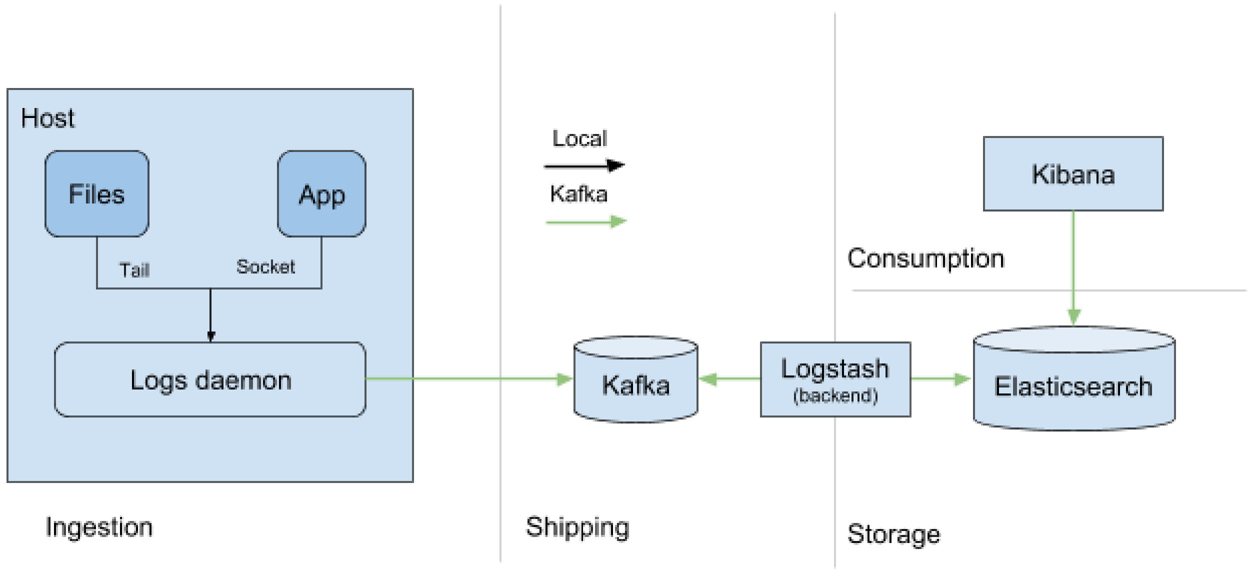
Non-goals

It is also important to outline what goals are out of scope for this document, both for reasons of brevity and focus of the document itself.

- Design and agree on a minimum common schema for logs (related to fields explosion problem mentioned later)
- Decide on "admission criteria" for logs to be in Logstash and thus accessible to people with an NDA signed.
- Address multi tenancy (related to the above, i.e. different access levels for different types of logs)

Overview

This section provides an overview of the major components and their relationships. We can talk about logging pipeline, or a composition of stages.



Logging pipeline inside a single data center

Ingestion

A log entry is accepted into the pipeline and will make its way to storage. Protocols and encoding vary between applications but can be abstracted into two broad categories: *structured* and *unstructured* to indicate whether the log is (essentially) a key -> value map (e.g. JSON logs) or opaque strings with limited metadata (i.e. syslog messages). Parsing logs into structured fields is a generic approach and is typically applied to logs from third party or legacy applications that don't support structured logging, for example network devices.

As of July 2018 ingestion happens by way of applications sending logs to Logstash in plain text over the network, with the majority of traffic using unreliable transports (i.e. UDP).

To improve ingestion reliability, security and simplify logging requirements for applications a local daemon is introduced. The daemon runs on every host and collects logs from applications, preferably via syslog or standard output and error via the system's journal.

Shipping

Once logs are ingested they need to be reliably shipped to storage. In the current infrastructure shipping over the network often happens via best effort transports like UDP. While UDP offers some benefits (low overhead) it is problematic because log events are often sent in the clear and may be lost in flight.

The shipping stage is centered around a queue for reliable and secure shipping. A queue decouples ingestion from storage, allowing producers and consumers to work at their own pace and easily manage sudden surges in logs production. For example an Elasticsearch cluster restart doesn't result in lost logs but in a temporary backlog stored in Kafka. The log daemon will talk directly to Kafka and produce JSON logs. Consumers will be in the backend, reading from the queue and writing to storage.

We have chosen to use Kafka as it is the de-facto queue in production at Wikimedia Foundation and it is well supported by the Elastic stack.

Storage

Elasticsearch is in use today for storing logs. It is something the Foundation has experience with in production and has worked well so far. Currently retention is 30 days due to resource constraints, mainly disk space. Historically, all logs have been funneled into a single index prefix. This practice is being changed to segregate different types of logs into different prefixes (e.g. syslog) and allows for example different retention policies for different indices. Using multiple indices also helps with the so-called "fields explosion" problem described in T180051.

As of July 2018 per-log storage requirements are on average as follows: (no replication)

`logstash`	~850 bytes
`logstash-syslog`	~450 bytes

When enrolling a new application these figures are useful to estimate the disk space requirements. For example the Linux kernel emits about 450k log lines per day across the fleet, resulting in 200MB per day required for storage (600MB, including 3x replication in a single site).

Consumption

Consumption in this is case refers to systems and use cases that sit downstream of the logging infrastructure. Further processing might be needed for the logs once they are shipped, to this end it should be simple to tap into the logs stream for example to calculate time series, store into HDFS for archival, browse via HTTP and so on.

Detailed design

Ingestion

The ingestion stage is by far the most important as far as applications are concerned: it is the interface through which logs are accepted and will be eventually indexed and stored. The proposal in this document is to have the following interfaces available: local syslog[1], standard output/error and *tailing* of log files, both handled by a *log daemon*. The former being preferred, and the latter to be used only in cases where logging to syslog proves to be more complex. For applications running as daemon standard output and error will also be captured by the system's journal and sent as part of the application logs.

As of July 2018 *rsyslog* is deployed across the fleet by virtue of being the standard Debian log daemon and carrying out multiple functions: send all syslog messages to centralized hosts[2], act as the central receiving server, and send applications logs (including MediaWiki) to Logstash via syslog over UDP. Applications not using local syslog for ingestion send their logs directly to Logstash using a variety of transports and encodings, whereas application logs written to disk are not ingested into Logstash or central syslog hosts.

As of today the proposed interface is implemented already for applications using syslog; applications sending logs directly to Logstash will need to be migrated either to syslog or log

[1] Over UNIX socket
[2] TLS encryption to be deployed to most of the fleet as per T136312

files (and their rotation). Introducing a log daemon between the application and Logstash also introduces various tradeoffs: on the plus side TLS is uniformly enforced by the log daemon (instead of the application) and logging requirements are simplified. However, these benefits come at the cost of increased overall complexity. Also note that due to the requirements outlined below and the fact that the log daemon will run on all hosts Logstash isn't a candidate for this task.

The syslog interface involves writing a syslog-formatted message (according to rfc5424) to a UNIX socket (the canonical location being /dev/log). The MSG field can contain a free-form string in case of unstructured logs or a JSON object for the structured case. In the latter case the JSON object will be parsed as such and sent to shipping, possibly augmented with metadata generated by the log daemon. In case of unstructured data the whole message will be wrapped into a JSON object with appropriate metadata and sent to shipping, the MSG field might undergo some basic parsing as needed.

Using a UNIX socket provides advantages compared to a TCP/UDP connection: access control happens based on the UID/GID of calling process, access to logging inside containers can happen by bind-mounting the socket from outside the container instead of requiring networking inside the container itself. One fundamental advantage is for the log daemon to receive information from the kernel about the sending process, allowing the log daemon to attach trusted metadata to log messages.

The log daemon will likely require per-application configuration, in the common case coming from Puppet. In the case of applications running on Kubernetes the log daemon configuration will come from *config maps* from Kubernetes itself.

Requirements for log daemon:

- Support multiple inputs: syslog, systemd journal, tailing log files
- Support output TLS, Kafka
- Rate limits for inputs
- Handling of long logs/events (e.g. stacktraces)
- Bonus: failover when specifying multiple outputs, support for json input, splitting logs into fields, spool to local files if outputs are unavailable

Type	Logs last 30d	Percent	Transport	Format	Library
mediawiki	1,668,972,724	70.78%	udp	json	monolog
parsoid	266,472,759	11.30%	udp	gelf	service-runner
wdqs	124,565,124	5.28%	udp	json+syslog	logback
ores	117,729,989	4.99%	udp	json	wsgi's logging
elasticsearch	77,215,960	3.27%	udp	gelf	log4j
logback	61,349,489	2.60%	udp	json	logback
syslog	11,520,139	0.49%	udp	syslog	proprietary, from network devices
restbase	8,601,422	0.36%	udp	gelf	service-runner
cpjobqueue	4,507,883	0.19%	udp	gelf	service-runner
parsoid-tests	3,482,283	0.15%	udp	gelf	service-runner
hhvm	3,199,859	0.14%	udp	syslog	custom, via rsyslog
webrequest	2,822,727	0.12%	tcp	json	via socat, from oxygen.eqiad
citoid	1,417,667	0.06%	udp	gelf	service-runner
eventstreams	1,409,819	0.06%	udp	gelf	service-runner
cassandra	1,354,945	0.06%	udp	json	logback

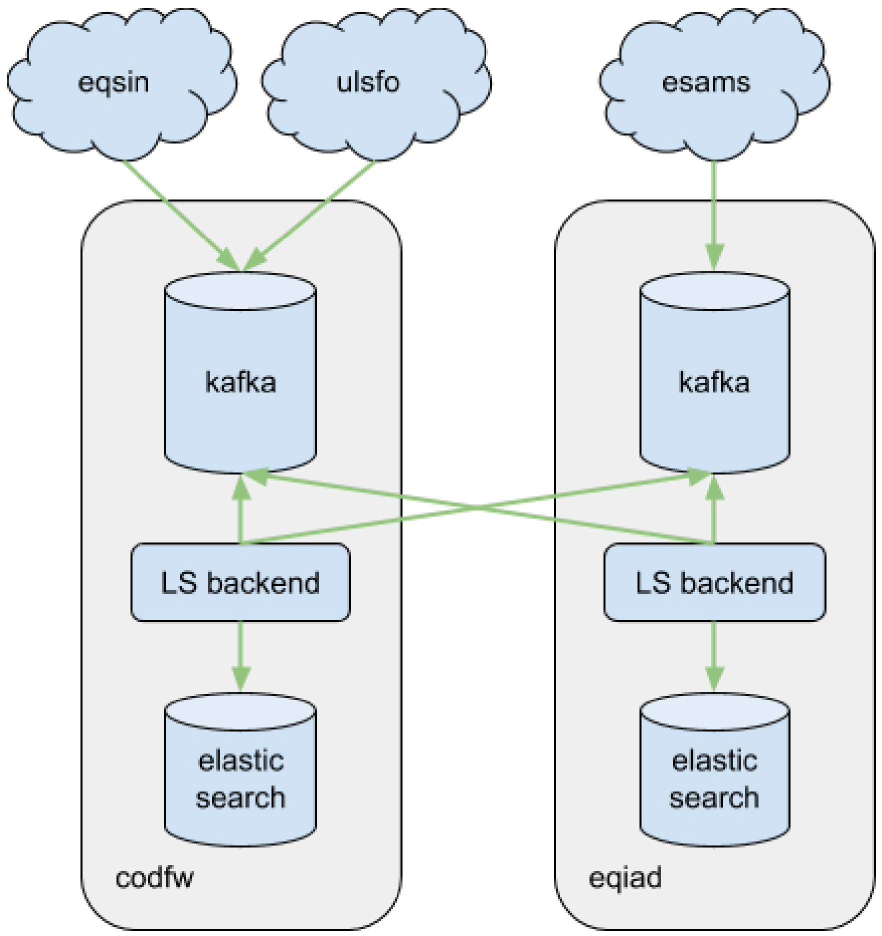
Top 15 logs producer over the last 30 days as of August 2018 (excluding long tail of 0.2% traffic)

One fundamental part of the logging pipeline is turning *unstructured* logs into *structured* ones: in other words converting logs from applications we don't control into meaningful key / value pairs. One powerful mechanism to accomplish this in Logstash is *grok*: a library of common regular expressions to extract fields.

For applications using unstructured logs grok parsing and splitting will happen on the Logstash backends when consuming from Kafka with the grok configuration managed by Puppet. To ensure maintenance and safety against regressions the grok configuration should be tested, ideally by continuous integration. To that end, applications should provide sample logs and their structured counterpart. Deploying new configuration will cause no loss of availability since Logstash supports hot-reloading configuration.

For applications already implementing their own parsing/splitting of logs (e.g. emitting logs from `varnishncsa`) no changes are needed, logs can be sent to syslog with JSON as the payload.

Shipping



Multi-datacenter configuration for the logging pipeline

The shipping stage consists of (in order):

1. Receiving logs over the network from the logging daemon
2. Producing logs to the Kafka queue
3. Consuming logs from the Kafka queue
4. Indexing and storage in Elasticsearch.

In this stage logs are shipped from each site to their closest Kafka queue (codfw or eqiad as of August 2018). Usage of TLS is mandatory when communicating with a Kafka queue that lives on a remote site (e.g. for caching sites).

As far as Kafka itself is concerned the production will happen on a dedicated topic per Kafka queue. Log rates received by Logstash as of July 2018 are in the order of 1500/s and expected to increase[3] as more applications are enrolled. The default Kafka topic retention should provide ample buffering margins, with an estimated 4G/hour (replication and compression excluded) required for Kafka storage.

Once logs are produced into the site-local Kafka queue they will be consumed by multiple Logstash backend instances. Redundancy in this tier is achieved through the Logstash Kafka plugin support of consumer groups. This enables other backend Logstash instances to take over if a consumer fails for some reason. Furthermore, logs are replicated to two sites. Each Logstash backend consumes from all other Kafka queue sites and writes to the local Elasticsearch. This provides log replication to two sites while avoiding the complexity of high latency multi-site clustering within Elasticsearch.

[3] Exact increase is unknown, though even a 10x increase shouldn't cause problems for Kafka.

Storage

Currently ELK data is stored on a 3 host bare metal Elasticsearch cluster hosted in the eqiad datacenter. The disk capacity of each Elasticsearch host ranges from 2.7TB to 3.4TB. Current utilization is at approximately 30% with ~865GB stored on each node (including Elasticsearch shard replication)

```
shards disk.indices disk.used disk.avail disk.total disk.percent host         ip           node
    78      867.6gb     868gb      2.6tb      3.4tb           24 10.64.48.109 10.64.48.109 logstash1006
    78      864.6gb     865gb      1.8tb      2.7tb           31 10.64.16.185 10.64.16.185 logstash1005
    79      864.3gb   864.7gb      1.8tb      2.7tb           31 10.64.0.162  10.64.0.162  logstash1004
```

In order to provide scale out capability in a seamless way, and because the number of differing hardware configurations backing Elasticsearch will continue to grow over time, we will introduce node metadata tags within Elasticsearch to differentiate hardware classes. We'll begin with two tags:

- SSD - High performance SSD backed hosts. Use case: indexing
- HDD - Medium performance rotating disk backed hosts. Use case: online search, longer-term storage

In order to make use of these tags we will modify the logstash Elasticsearch template with a routing allocation requirement that directs newly created indices to the currently appropriate storage type for indexing. Then, as indices age, a curator job will alter the allocation. This instructs Elasticsearch to (seamlessly and online) migrate data shards to another storage class.

The life of a logstash index:

- Initial index creation occurs on SSD indexing tier
- After 7 days index is moved to HDD online search tier

Since identical data will be stored in both eqiad and codfw, we plan to support depooling of a single site for maintenance. Likewise, we plan to support resynchronization after an outage resulting in ELK data loss at a single site.

High level site depooling procedure:

1. Reindex .kibana index from the active cluster into the to-be-promoted cluster via Elasticsearch remote reindex
2. Redirect Kibana traffic to the desired Kibana frontend (via varnish puppet commit)
3. Redirect log daemon traffic to the desired Kafka cluster hosted in the to-be-promoted site (ideally automatically handled by logging daemon, alternatively performed manually via dns)

High level data recovery/re-synchronization procedure:

1. Depool affected site (see above)
2. Address issue(s) impacting availability at affected site
3. Ensure service is restored at affected site and allow affected site to ingest data for 24 hours while remaining depooled.
4. Recover any indices with data gaps from the working site using the reindex from remote api https://www.elastic.co/guide/en/elasticsearch/reference/current/docs-reindex.html#reindex-from-remote

Consumption

There are two high level categories of ELK consumers -- Tooling and users.

Users consume log data through Kibana. Kibana is a web interface that provides log search, visualizations (charts, graphs, etc.) and dashboarding functionality (groupings of saved searches and visualizations). Kibana runs in a HA configuration, and speaks directly to the Elasticsearch cluster. We currently provide Kibana access to users with membership to LDAP groups ops, nda and wmf. There is no plan at the present time to modify this (see non-goals section)

Tooling generally consumes Logstash log data directly from Elasticsearch. Because of this it is important that we prepare to provide connectivity for tools to access the API of the Elasticsearch cluster storing ELK log data. There are no plans in this document to introduce specific new tools that consume log data, though such tools would be useful. As an example of the type of tool being described is Elastalert. Elastalert is software which performs queries against Logstash data in Elasticsearch on a regular interval and generates alerts based on the output. Another example is go get my logs used to provide a familiar command line interface to fetching logs.

Alternatives

The main interface to logging is writing JSON messages to Kafka, with local log daemon coming second. Applications able to write JSON to Kafka are encouraged to do so; communications to Kafka are required to use TLS when crossing datacenter boundaries and TLS is recommended when a Kafka cluster exists inside the datacenter.

For applications that can't write to Kafka, sending JSON with syslog header to the log daemon over UNIX socket is the recommended alternative for structured messages. For applications running as daemons standard output and error will be also captured and sent as part of the application's logs.

For third party applications that don't support structured logging, unstructured syslog is the recommended interface. Such messages will be regarded as generic syslog messages and sent as JSON to Kafka by the log daemon. Further splitting/processing can happen when Logstash backend consumes from Kafka and before storage.

Cross cutting concerns

This section outlines considerations common to all services SRE operates, as such are not included in the main design.

Privacy and security

- TLS across the board for communications over the network
    - Logs daemon can use the host certificate from puppet
    - For server certs we can use certgen-issued certs
- When ingestion is local we can ask the kernel for the UID of the running process and attach that to the log
- For the future: SIEM, log sealing, auditing, etc

Observability

- Most, ideally all, components have Prometheus metrics exposing the “golden signals” below
- **Latency** time from log ingestion to being able to query the ingested log
- **Traffic** logs/s and bytes/s (ingested/shipped/stored), number of Elasticsearch queries running
- **Errors** logs/s dropped (not ingested/shipped/stored), Elasticsearch cluster status (red/yellow)
- **Saturation** queues utilization (both internal e.g. to Logstash or external like Kafka)
- Grafana dashboards for the whole logging pipeline (via four golden signals)
- Alerts for thresholds based on golden signals

Backups

Backing up log data is not being considered in this document, given the retention guidelines and the significant size requirements. Kibana dashboards will need periodic backups, likely via elasticsearch-dump and storage in WMF’s backup infrastructure.

Scalability

We should be able to seamlessly add capacity at each layer with no downtime, and without requiring major config changes.

We should prepare for differing hardware classes (e.g. high performance indexing and lower performance archival) and ensure we have the ability to seamlessly migrate between them.

We should document estimated upper boundaries at each layer and set up mechanisms to provide actionable warnings with adequate advance notice.

Availability

One of the main tenets of SREs is achieving the expected availability, and the feature #1 of any system is that *it works*.

Ingestion availability depends on the local log daemon working as intended. Namely the daemon must be running, accepting logs from local applications and tailing log files. A non-running daemon will cause unavailability for applications using syslog and no unavailability for applications writing to log files. When the log daemon is down particular care must be taken to ensure applications gracefully handle blocking writes to syslog. Kafka is involved in received collected logs over the network: its availability depends on the network and hosts running Kafka itself.

Maintenance to Kafka should result in no loss of availability as long as enough brokers are available.

Shipping involves producing to the closest datacenter Kafka queue, and Logstash backend consuming from all queues. Upon Kafka unavailability the producers should spool logs locally until Kafka is available again, and consumers will pick up where they left off. In other words Kafka unavailability resulting in a slow down of the logging pipeline with properly-behaved producers.

Storage availability is bound to Elasticsearch availability. While Elasticsearch clusters are designed to be highly available certain operations like major version upgrades require a full cluster restart. When it isn't possible to write to Elasticsearch the Kafka consumers can either spool locally or stop consuming from Kafka, and resume once Kafka is available again.

Consumption is equally bound to Elasticsearch: upon unavailability of Elasticsearch consumption (e.g. Kibana) can't work. To mitigate this kind of unavailability and assuming independent Elasticsearch clusters fail independently, consumption can be switched to an available Elasticsearch cluster.

Kibana currently is hosted on a single VM, though its availability can be increased with little (same datacenter) or moderate (different datacenters) effort. For example adding more VMs in diverse datacenters and making sure the dashboards storage is replicated.

Maintenance

We should be able to take down individual components for maintenance without significant impact overall.

Ingestion maintenance involves upgrading the logs daemon, causing temporary unavailability for applications sending logs to the daemon, and no unavailability when tailing log files.

Shipping Kafka maintenance should cause no lost logs; brokers can be rebooted individually and clients will fail over to other brokers. Kafka maintenance should result in no lost logs; the log daemon will spool logs locally and retry shipment once Kafka comes back up. Taking down Logstash backends one at a time will cause no problem because other Logstash backends will take over consuming from Kafka.

Storage Elasticsearch maintenance or brief unavailability should cause no lost logs, Kafka consumers will stop and resume writing logs when Elasticsearch is available again.

Consumption maintenance shouldn't impact the infrastructure in any way because it sits on the read path. For stateless cases like Kibana talking to the same Elasticsearch cluster replication on multiple hosts is trivial.

Deprecation

This section outlines some scenarios for deprecation of components used by the logging pipeline, the rationale being that when setting up a new system it is also important to consider how to tear such system down.

Ingestion deprecation means swapping the logs daemon with another one, the daemon itself is stateless and thus deprecation should be straightforward.

Shipping deprecation as far as Kafka is concerned is possible to replace with a similar queue, though unlikely to happen since Kafka has proven reliable in production. Replacing Logstash is also possible because any Kafka consumer will do. Parsing unstructured logs into structured ones can also be deprecated, most likely moved to the log daemon and thereby distributing the load from Logstash backend onto individual hosts. Another possibility for parsing unstructured logs is so-called stream processing: in this case consuming from Kafka, transform/parse as needed and produce back to Kafka.

Storage deprecation will entail changing Elasticsearch for something else with similar functionality, while possible it is fair to say it is pretty unlikely to happen in the next five years.

Consumption specifically Kibana, the same considerations for Elasticsearch apply: possible but unlikely at least for the next five years.