Verwalter Documentation

Release 0.13.4

Paul Colomiets

Aug 10, 2018

Contents

Contents:

CHAPTER 1

About

Contents:

1.1 Concepts

Verwalter is a cluster orchestration tool

Briefly verwalter does the following:

- Starts configured set of services
- Monitors cluster load and changes number of workers on demand
- Does gradual software update of supervised services triggered by operator
- Provides limited form of service discovery
- All the features are scriptable by clean and simple Lua code fragments

It builds on top of lithos (which is isolation, containerization, and supervising service) and cantal (which is sub-real-time monitoring and node discovery service).

Verwalter is a framework for long-running services. It has abstractions to configure running 10 instances of service X or use 7% of capacity for service Y. The resources are consumed until configuration changed. Contrast this approach with Mesos or Yarn which has "start task A until it completes" abstraction. (However, Verwalter can run and scale Mesos or Yarn cluster).

1.1.1 Components

Let's look through each component of the system first. This is very helpful to understand the big picture outlined below.

Note the setup of the cluster is flat: you need all three components `verwalter`, `lithos` and `cantal` on all nodes.

Lithos

Lithos is essentially a process supervisor. Here is the basic workflow:

1. Read configuration at `/etc/lithos/sandboxes`
2. For each sandbox read configuration in `/etc/lithos/processes`
3. Prepare the sandbox a/k/a linux container
4. Start process and keep restarting if that fails
5. Add/remove process if configuration changed

Lithos provides all necessary isolation for running processes (except it does not handle network at the moment of writing), but it's super-simple comparing to docker and mesos (i.e. mesos-slave) and even systemd:

- Lithos reads configuration from files, no network calls needed (note the security impact)
- Lithos can restart itself in place, keeping track of processes, so it's mostly crash-proof
- On `SIGHUP` signal (configuration change) it just restarts itself

The **security model** of lithos is the ground for the security of whole verwalter-based cluster. So let's take a look:

- It's expected that configs for `sandboxes` are predefined by administrators and are not dynamically changed (either by verwalter or any other tool)
- Sandbox config constrains folders, users, and few other things that an application can't escape
- The command-line to run in sandbox is defined in image for the application

All this means that verwalter can only change the following things:

- Image (i.e. version of image) to run command from
- The name of the command to run from limited set of options
- Number of processes to run

I.e. whatever evil would be in verwalter's script it can't run an arbitrary command line on any host. So can't install a rootkit, steal users' passwords and do any other harm except taking down the cluster (which is an expected permission for resource scheduler). This is in contrast to docker/swarm and mesos that allow to run anything.

Cantal

The cantal is a semi-real-time monitoring tool. It delivers statistics in unusually short intervals and provides *node* discovery.

We use it:

- As a node discovery and availability monitoring
- For looking at current metrics of started application in nearly real-time
- As a liveness check for applications (mostly by looking at metrics)
- For collecting metrics from all nodes and aggregating
- For fetching limited amount of historical data (~1 hour)

Verwalter

The verwalter is final piece of the puzzle to build fully working and auto-rebalancing cluster.

In particular it does the following:

1. Establishes leader of the cluster (or a subcluster in case of split-brain)
2. Leader runs model of the cluster defined by sysadmin and augmented with lua scripts, to get a number of processes run at each machine (and other important pieces of configuration).
3. Leader delivers configuration to every other node
4. At every node, the configuration is rendered to local configuration files (most importantly `/etc/lithos/processes`, but other types of configuration are supported too), and respective processes are notified.
5. All nodes display web frontend to review configuration. Frontend also has actionable buttons for common maintainance tasks like software upgrade or remove node from a cluster

Unlike popular combinations of etcd + confd, consul + consul-template, or mesos with whatever framework, verwalter can do scheduling decisions in split-brain scenario even in minority partition. Verwalter is not a database so having two leaders is not a problem when used wisely.

Note: Yes you can control how small cluster must be for cluster model to work, and you can configure different reactions in majority and minority partition. I.e. doing any decisions on a single node isolated from 1000 other nodes is useless. But switching off external *memcache* instance for the sake of running local one may be super-useful if you have a micro-service running on just two nodes.

The Missing Parts

In the current implementation, the missing part of the puzzle is a means to deliver files to each box. In particular the following files might need to be distributed between nodes:

1. Images of containers for lithos
2. Vervalter's configs and configuration templates

We use ansible and good old rsync for these things for now

1.1.2 The Big Picture

The cluster setup is simple. We have only one type of node and that node runs three lightweight processes: lithos, cantal and verwalter.

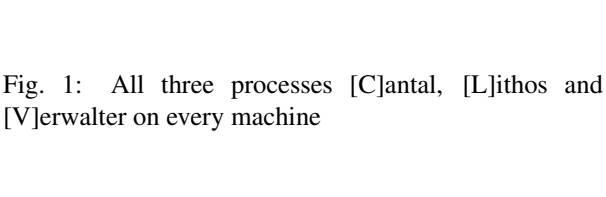

Fig. 1: All three processes [C]antal, [L]ithos and [V]erwalter on every machine

As outlined above cantal does node discovery by UDP. When the node first time becomes up, it needs to join the cluster. Joining the cluster is done by issuing a request:

```
curl http://some.known.host:22682/add_host.json -d '{"addr": "1.2.3.4:22682"}'
```

Warning: This is not a stable API, so it may change at any time.

Fig. 2: Propagation of cluster join message

As the nodes are all equal you can issue a request to any node, or you can add any existing node of a cluster to the new node, it doesn't matter. All the info will quickly propagate to other nodes via gossip protocol.

As illustrated on the picture the discovery is random. But it tuned well to efficiently cover the whole network.

When starting up, verwalter requests cluster information **from local cantal instance**. The information consists of:

- list of peers in the cluster
- availability of the nodes (i.e. time of last successful ping)
- some minor useful info like round trip time (RTT) between nodes

Fig. 3: Initial request of cluster info

Verwalter delegates all the work of joining cluster to cantal.

As described above, verwalter operates in one of the two modes: leader and follower. It starts as follower and waits until it will be reached by a leader. The Leader in turn discovers followers through cantal. I.e. it assumes that every cantal that joins the cluster has a verwalter instance.

Note: While cantal is joining cluster and verwalter does its own bootrapping and possible leader election, the lithos continues to run. The above means if there was any configuration for lithos before a reboot of the system or before you do any maintenance of the verwalter/consul, the processes are started and supervised. Any processes that crash are restarted and so on.

In case you don't want processes to start on boot, you may configure the system to clean lithos configs on reboot (for example by putting them on `tmpfs` filesystem). Such configuration is occasionally useful, but we consider the default behaviour to start all processes that were previously run more useful in most cases.

Leader's Job

When verwalter follower is not reached by a leader for the predefined time (don't matter whether it is on startup or after it had a leader), it starts an election process. The election process is not described in detail here because it's work in progress. It will be described in detail later in other parts of documentation.

When verwalter elected as a leader:

1. It connects to every node and ensures that every follower knows the leader
2. After establishing connections, it gathers the configuration of all currently running processes on every node
3. It connects to local cantal and requests statistics for all the nodes
4. Then it runs scheduling algorithm that produces new configuration for every node
5. At next step it delivers configuration to respective nodes
6. Repeat from step 3 at regular intervals (~10 sec)

In fact, steps 1-3 are done simultaneously. As outlined in cantal documentation it gathers and aggregates metrics by itself, easing the work for verwalter.

Note that at the moment when a new leader is elected the previous one is probably not accessible (or there were two of them, so no shared consistent configuration exists). So it is important to gather all current node configurations to keep

number of reallocations/movements of processes between machines at a minimum. It also allows to have persistent processes (i.e. processes that store data on the local filesystem or in local memory, for example, database shards).

Having not only old configuration but also statistics is crucial, we can use it for the following things:

1. Detect failing processes
2. Find out the number of requests that are processed per second
3. Predict trends, i.e. whether traffic is going up or down

All this info is gathered continuously and asynchronously. Nodes come and leave at every occasion, so it is too complex to reason about them in a reactive manner. So from SysOp's point of view the scheduler is a pure function from a {*set of currently running processes*; *set of metrics*} to the new configuration. The verwalter itself does all heavy lifting of keeping all nodes in contact, synchronizing changes, etc.

The input to the function in simplified human-readable form looks like the following:

```
box1 django: 3 running, 10 requests per second and growing; 80% CPU usage
box2 flask: 1 running, 7 RPS and declining; django: 2 starting; 20 %CPU
```

In lua code function looks like this (simplified):

```
function scheduler (processes, metrics)
    ...
    return config
end
```

Furthermore, we have helper utilities to actually keep matching processes running. So in many simple cases scheduler may just return the number of processes it wants to run or keep running. In simplified form it looks like this:

```
function schedule_simple(metrics)
    cfg = {
        django_workers = metrics.django.rps / DJANGO_WORKER_CAPACITY,
        flask_workers = metrics.flask.rps / FLASK_WORKER_CAPACITY,
    }
    total = cfg.django_workers + cfg.flask_workers
    if total > MAX_WORKERS then
        -- not enough capacity, but do our best
        cfg = distribute_fairly(cfg)
    else
        -- have some spare capacity for background tasks
        cfg.background_workers = MAX_WORKERS - total
    end
    return cfg
end

make_scheduler(schedule_simple, {
    worker_grow_rate: '5 processes per second',  -- start processes quickly
    worker_decline_rate: '1 process per second', -- but stop at slower rate
})
```

Of course the example is oversimplified, it is only here to get some spirit of what scheduling might look like.

By using proper lua sandbox, we ensure that function is *pure* (have no side effects), so if you need some external data, it must be provided to cantal or verwalter by implementing their API. In lua script, we do our best to ensure that function is idempotent, so we can log all the data and resulting configuration for **post mortem debugging**.

Also this allows us to make "shadow" schedulers. I. e. ones that have no real scheduling abilities, but are run on every occasion. The feature might be useful to evaluate new scheduling algorithm before putting one in production.

Follower's Job

The follower is much simpler. When leadership is established, it receives configuration updates from the leader. Configuration may consist of:

1. Application name and number of processes to run
2. Host name to IP address mapping to provide for an application
3. Arbitrary key-value pairs that are needed for configuring application
4. (Parts of) configurations of other nodes

Note the items (1), (4) and partially (3) do provide the **limited form of service discovery** that was declared at start of this guide. The (2) is there mostly for legacy applications which does not support service discovery. The (4) is mostly for proxy servers that need a list of backends, instead of having backends discover them by host name.

Note: We use extremely ignorant description of "legacy" here. Because even in 2015 most services don't support service discovery out of the box and most proxies have a list of backends in the config. I mean not just old services that are still widely used. But also services that are created in recent years. Which is problem on it's own but not the one verwalter is aimed to solve. It's just designed to work both with good and old-style services.

Every configuration update is applied by verwalter locally. In the simplest form it means:

1. Render textual templates into temporary file(s)
2. Run configuration checker for application
3. Atomically move configuration file or directory to the right place
4. Signal the application to reload configuration

For some applications it might be more complex. For lithos which is the most common configuration target for verwalter it's just a matter of writing YAML/JSON config to temporary location and calling `lithos_switch` utility.

Note: We're still evaluating whether it's good idea to support plugins for complicated configuration scenarios. Or whether the files are universal transport and you just want to implement daemon on it's own if you want some out of scope stuff. The common case might be making API calls instead of reloading configuration like you might need for docker or any cloud provider. Lua scripting at this stage is also an option being considered.

1.1.3 Cross Data Center

When crossing data center things start to be more complicated. In particular verwalter assumes:

Fig. 4: The cross data center connection scheme

1. Links between data centers are order of magnitude slower than inside (normal RTT between nodes inside datacenter is 1ms; whereas between DC even on the same continent 40ms is expected value and sometimes may be up to 120-500 ms). In some cases traffic is expensive.
2. The connection between datacenters is less reliable and when it's down clients might be serviced by single data center too. It should be possible to configure partial degradation.
3. Each DC has some spare capacity on it's own. So moving resources between data centers might be more gradual.

4. There are few data centers (i.e. it's normal to have 100-1000 nodes, but almost nobody has more than a dozen of DCs).

So verwalter establishes a leader inside every datacenter. On the cross-data-center boundary all verwalter leaders treated equally. They form full mesh of connections. And when one of them experiences peak load it just requests some resources from other.

Let's repeat that again: because verwalter is not a database, consistency is not important here. I.e. if some resources are provided by DC1 for DC2 and for some reason latter lost connectivity or has some other reason to not use requested resources, we just release them on a timeout by looking at appropriate metrics. So dialog between data center leaders translated to the human language may look like the following:

All things here are scriptable. So your logic may only move background tasks across data-centers or use cloud API's to request more virtual machines

---

Note: A quick note to last sentence. You can't access cloud API directly because of sandboxing. But you may produce a configuration for some imaginary *cloud provider management daemon* that includes bigger value in the setting *number of virtual machines to provision*.

1.2 Glossary

configuration The initial input to the verwalter's scheduler. It conists of:

- All data in `/etc/verwalter/runtime/*`
- All templates and actions in `/etc/verwalter/templates/*`

It's expected that these files are never mutated. But new ones might be added. E.g. if there is `runtime/v1.0.1/..` and new version `runtime/v1.0.2` appears verwalter reads it as fast as possible, and makes it available on next scheduler run.

All configuration versions are read by verwalter. So you can write any required logic in scheduler. For example, to arrange a blue/green deployment strategy you may need to keep "blue" configuration around even when no processes running it are present.

schedule A data structure that holds information about all the services that must run on the whole cluster. This is the result of running a scheduler code.

In fact it's just a piece of JSON-like data, which you may use in templates when rendering the configurations. It may contain anything, but usually it's something along lines of nested dicts: `host-name -> process-name -> number-of-instances`.

scheduler The Lua code that receives a *configuration* and a *state* and generates a *schedule*. Basically it's just a (pure) function.

A scheduler may do whatever it needs for the transformation. But, but it's very important to obey the following rules:

1. No external data should be used. Just *configuration* and *state*.
2. No side effects allowed, like writing to the files or even reading current date/time (we provide date/time as part of state, though)
3. It shouldn't be too slow

deployment id The unique identifier of the series of the actions that was run to apply certain config. Deployment id is local for single machine, but may span across roles. Single deployment id is used only once, so they refer to the time range when deployment started and finished. Multiple deployments can't be run on single machines simultaneously.

Not all roles can be deployed with the single deployment id just the ones which need an update. Each role may execute commands only once during single deployment.

There is no direct correspondence between config hash and deployment id. Single config may be deployed multiple times even on single machine. (each time when verwalter is restarted, each time when config changed and then rolled back again). But single deployment may deploy only single configuration. I.e. configuration can't change during deployment.

And there is no direct match between application update and deployment id. The (rolling) application id usually involves multiple configuration updates. And each configuration update triggers one deployment on each machine. Also multiple rolling updates of different applications may take place at the same time. And all of them correspond to a single configuration change at any point in time.

role A single deployment unit. A role has it's own configuration independent of others(set of versions of containers, set of config templates).

A role may contain multiple containers. And multiple different setups on different nodes. It's up to a lua configuration.

Usually single role refers to single "sandbox" in lithos, but this limit is not enforced.

Similarly blue/green deploy (or rolling update) between versions is usually performed for a role. Which means each role has it's own state of the deployment, and multiple roles can be migrated independently. But this is not enforced either. With careful scripting you can do both: synchronize updates of multiple roles or update different processes in single role using some independent states.

CHAPTER 2

Running

Contents:

2.1 Exit Codes

2.1.1 Verwalter Daemon

- `3` – initial configuration read failed
- `4` – failed to load scheduler's lua code
- `5` – failed to add inotify watch
- `81` – internal bug: tcp listener exited
- `82` – internal bug: fetch channel is dead
- `83` – internal bug: responder thread is dead
- `91` – killed by watchdog of scheduler, which means:
  - scheduler has not finished it's work within one second
  - scheduler lua scripts could not be initialized within ten seconds
  - "runtime" metadata could not be loaded within 2 seconds
  - inotify continuously reports changes during 10 seconds
- `92` – scheduler thread have panicked (probaby a bug)
- `93` – killed by watchdog of the render/apply code. This probably means either your templates are a way too slow, or commands that are used to apply config are doing too much work. We currently have a fixed timeout of 180 seconds (3 min) for all of the stuff there (normally it's done in a fraction of second)
- `94` – the thread that applies config have panicked (probably a bug)
- `95` – no leader was elected for last 5 min

2.1.2 Verwalter Render

This may be visible in verwalter's deployment log:

- `2` – argparse error, should not happen, but may be if version of verwalter-render (on disk) doesn't match verwalter daemon running
- `3` – error validating arguments, should be treated same as `2`
- `4` – no `template` key found in metadata, this means scheduler returned incomplete data for this role
- `5` – verwalter daemon is running different version from verwalter-render. This probably means you should restart verwalter daemon. For other things it should be treated same as `2`
- `10` – error when reading or rendering templates
- `20` – error appling templates (executing commands)
- `81` – error when doing logging, this probably means that some errors are absent in logs

The error codes marked with mean that no actual rendering process is started. I.e. system is consistent (old) state. With other codes we can't easily say whether configuration was appllied partial, comprehensively or not at all.

2.2 Configuration Directory Layout

The layout of /etc/verwalter directory.

The directory layout is still in flux. Here are somewhat current draft.

- `scheduler` – scheduler code in lua
  - `scheduler/SCHEDULER_VERSION/main.lua` – the entry point of the scheduler (`scheduler` function)[1]
  - `scheduler/SCHEDULER_VERSION/**/*.lua` – other files that are `require`'d from scheduler
- `templates` – the templates to render configuration locally
  - `templates/ROLE/TMPL_VERSION` – templates for *role* and version[1]
    - `**/*.hbs` – bare configuration templates
    - `**/*.vw.yaml` – instructions on how to apply the template
- `runtime` – the runtime metadata, mostly list of processes to run and other data needed for scheduling. Basically all of this is passed to the scheduler
  - `runtime/ROLE/ROLE_VERSION` – metadata dir for *role* and version
    - `NAME.yaml` – adds some metadata under key `NAME`
    - `NAME.json` – just another format of the same thing
- `machine` – the current machine metadata
  - `NAME.yaml`, `NAME.json` – adds some metadata under key `NAME`
- `frontend` – the files to render the frontend[2]

[1] The version of scheduler and version of templates is not the same as version of *role* (i.e. an application). It's expected that scheduler and templates change very rarely and only by admins, not by release managers. Also you might use "shadow" scheduler and "shadow" template renderer for debugging.

[2] Each installation have different needs. So verwalter doesn't have a frontend that is packaged with verwalter. We only provide the API, and a default (or example) frontend which you might use as a starting point. Sure verwalter serves static files so you don't need to install a separate web server.

  - `common/*` – common files for the whole cluster (e.g. libraries)
  - `ROLE/*` – role-specific things[3]
- `sandbox` – this contains some security configs:
  - Logs that can be served within verwalter
  - (TODO) commands to run from verwalter-render, run-as user, etc.

Note: We avoid the term "application" here because it's inherently vague. The *role* is just unit that may be deployed independently (so it's also versioned independently). The role may consists multiple applications or application may be built on top of multiple roles, dependening on use case and how you define the application.

2.2.1 Deployment

It's assumed that `scheduler` and `templates` are written by SysOps. They should be versioned in version control system and deployed as needed.

The `frontend` is very similar. It should be versioned too. It's only mentioned separately because usually changed by some frontender or release engineer or whatever.

The `runtime` folder is assumed to be deployed by buildbot. I.e. when build is done, buildbot does two things to prepare deployment:

1. Upload built image to all servers that will be able to run the application
2. Put app metadata in the `runtime` folder on same machines

Then it's up to the scheduler if it deploys the version automatically or waits for operator to trigger the update action.

[3] We don't have frontend files versioned yet. It's not critical part of the system and it assumed that an (updated) frontend should support at least few versions of the application (*role*).

CHAPTER 3

Writing Scheduler

Contents:

3.1 Scheduler API

3.1.1 Overview

> **Warning:** API is still unstable and is subject to change

Scheduler is a lua script. All the API are exposed through functions on the main module.

3.1.2 Callbacks

Functions that verwalter calls on its own.

> **Note:** You can use coroutines inside the code, but you can't `yield` to rust code. I.e. the code is always synchronous and must return the value on each call. However, you can store some custom state in the schedule itself.

`schedule`(*named_arguments*)

Arguments

- **`peers`** – List of peers and pings to them as reported by cantal
- **`runtime`** – Metadata stored in */etc/verwalter/runtime*
- **`parents`** – List of parent schedules (the ones that are active now). Usually there is only one. But when we join cluster just after split-brain there can be more than one parent schedule

- **metrics** – Metrics as returned by cantal

Return value of the scheduler is a JSON object with the following keys:

vars Mapping (json object) that contains arbitrary variables which will be passed to the renderer. They might be overriden by role and node-specific variables. See below.

Example:

```
{"vars": {
    "cluster_name": "dev"}}
```

roles Mapping of role to vars of this role. This contains variables common for specific role on all nodes. All roles specified here will be rendered on all machines (can spawn `0` instances, though).

Example:

```
{"roles": {
    "django": {
        "version": "v0.1.3",
        "listen-ports": "8080"}}
```

nodes Mapping of node name (short/unqualified hostname) to node metadata. Each node contains: *vars* and *roles*.

Example:

```
{"nodes": {
    "alpha": {
        "vars": {"nearest_cache_addr": "slave7.redis.local"},
        "roles": {
            "django": {
                "instances": 1,
                "version": "v0.1.3"}}}}}
```

More information on how variables are composed for the renderer is in *Rendering* docs.

query_metries A query for metrics. It's sent directly to cantal. Refer to cantal's documentation to find out the structure of the metrics.

3.2 Rendering

In verwalter "rendering" is a process of applying schedule to configure specific application. It may consist of:

1. Substituting variables in textual templates
2. Running shell commands
3. Sending signals to other processes or different kind of IPC
4. *Possible, but discouraged:* calling HTTP APIs

Rendering for every role is deemed to be indepenedent of other roles. We also encourage, but cannot enforce the following properties:

1. Atomic render of role (i.e. either it applied entirely, or not at all)
2. Full configuration check before switching

3.2.1 Input

Input to the rendering process is a mapping of variables to values. For each role we merge the following items from schedule:

- `vars`
- `roles[role_name]`
- `nodes[node_name]["vars"]`
- `nodes[node_name]["roles"][role_name]`

Where latter variables override former ones.

Nested mappings are merged up to two level's deep. I.e. if `vars["common"]` is a mapping each key of it will be updated by `roles[x]["vars"]["common"]` independently, but `vars["common"]["info"]` would be replaced as a single atomic unit, regardless of whether it is an object or a string.

CHAPTER 4

Writing Renderers

Renderer is a subsystem that executes all necessary steps to apply verwalter schedule/configuration on each machine

Contents:

4.1 Render Commands

4.1.1 Condition

Condition is a special command that executes other commands only if some condition happens.

Example:

Conditions:

`dirs-changed` Calculates hash of all files in the directory recursively at the beginning of the **processing this .render.yaml** file. Then the hashsum is checked again when `!Condition` is encountered and if hashsum changed `commands` are executed, otherwise they are silently skipped.

Options:

`commands` List of commands to execute when condition is true. All the same commands suported except the `!Condition` itself.

4.1.2 CleanFiles

Cleans files by pattern, keeping only ones listed.

Example:

Options:

`pattern` Filename pattern to check. This supports basic **glob** syntax plus any part of path can be *captured* like in regular expression. This means that only parenthised part is matched against keep list, and only files that match glob are removed.

Few pattern examples:

- `/dir/(*).conf` deletes `*.conf` files, `keep-list` contains file names without extension
- `/dir/(*.conf)`, same but `keep-list` contains filenames with extension
- `/dir/(**/*.conf)`, deletes `*.conf` recursively, where keep list contains relative path (*without* `./`)

`keep-list` Filename of the file which lists **names** which should be kept. Each line represents single name. The contents of each line matched against thing captured in `pattern` (see above). No comments or escaping is supported, empty lines are ignored.

CHAPTER 5

Tutorial Deployment

Warning: This is a work in progress tutorial for work in progress tools. It's not ready for use yet.

5.1 Brief

This tutorial will guide you though deploying simple django application using vagga, lithos, cantal and verwalter.

5.1.1 Tools

We are trying to assume as little as possible about the reader knowledge, but basic understanding of unix is definitely required. Here is the description of tools that most readers would be intoduced here to:

vagga A tool for setting up development environments. For this tutorial, we will use it for building container images. Similar tools: vagrant, docker-compose, otto, packer (in some sense).

lithos A container supervisor. This one starts containers in production environment. Unlike docker it doesn't have tools for building and fetching container images we will use vagga and rsync for that tasks. Similar tools: docker, rocket, systemd-nspawn.

cantal A monitoring system, or a system collecting statistics. It's main distinction is that it is decentralized. It stores data in memory, and keeps only recent data. This makes it fast and highly-available. And this in turn allows to make orchestration decisions based on the metrics. Another feature is that it has built-in peer discovery. Similar tools: collectd, prometheus, graphite.

verwalter A orchestration system. It's highly scriptable and decentralized. Meaning you can do orchestration tasks in split-brain scenario and it depends on you what specific things system can actually do. The tool also includes text templates for rendering configuration for any external system that is included in the cluster. Similar tools: mesos, kubernetes.

Any tool can potentially replaced by some other tool. Currently, the only hard dependency is that you need cantal to run verwalter.

Anyway this combination provides good robustness, security and ease of use. See *Concepts* for more details about how these tools rely on each other to provide mentioned features.

5.2 Container

Usually you start with a vagga container that works locally. There is a tutorial for building a container for django application. We will skip this part and assume you have a working container. Please, don't skip this part even if you have development environment already set up (but not containerized). It is important for the following reasons:

1. You need to know all dependencies and their versions, in may happen that you don't know exact list of system dependencies if you are using virtualenv for example.
2. Vagga makes everything readonly by default, so as lithos. This serves as additional check of which filesystem paths are writable by the application (hopefully you don't have any).
3. We'll need the container for the next steps. We will base our deployment container on the development one (see below)

It's also good idea to make add a check of whether your application needs a writable `/tmp`. Just add a volume to your vagga container config:

```
containers:
  django:
    ...
    volumes:
      /tmp: !Empty
```

This makes `/tmp` read-only. So you can see errors when application tries to write there and either fix the application (preferred in my opinion) or provide valid `/tmp` mount in lithos configs later on.

5.3 Preparing Machines

As described in *concepts*, you need to install lithos, cantal and verwalter on all three machines.

(TBD: we skip exact installation instructions for now, because we don't have repositories online yet).

5.3.1 Global Things

Verwalter (and cantal too) requires `/etc/machine-id`. If your system is running by systemd then you already have this file. Otherwise, you can either use `systemd-machine-id-setup` from systemd utilities, or just run simpler script like `uuidgen | sed s/-//g > /etc/machine-id`. You must run the script once on every machine and file must never change. **Don't put the file in the virtual machine image such as AMI**. System will malfunction if several machines have same machine-id.

5.3.2 Lithos Configuration

Here is a checklist:

1. `/etc/lithos/master.yaml` (doc) – might be empty but can be present
2. `/etc/lithos/sandboxes/APP_NAME.yaml` (doc) – must be present for each application, you want to deploy on the machine

3. `/etc/init/lithos.yaml` or `/usr/lib/systemd/system/lithos.service` should start `lithos_tree` daemon

These configs are not generated by verwalter for security reasons. For example, sandbox config limits the directories on a host system that application is able to read or write. We don't want any application that can reach verwalter's HTTP API to be able to change such fundamental constraints.

On the other hand, the reasons above doesn't tell you can't automate deploying these files. You can easily use ansible to upload them or put them into virtual machine image, such as AMI.

5.3.3 Cantal Configuration

CHAPTER 6

Verwalter Changes by Version

6.1 Verwalter 0.13.4

- Feature: log of invoked actions added with logger `verwalter::frontend::api::actions`

6.2 Verwalter 0.13.3

- bugfix: fix displaying actions on leader using default frontend

6.3 Verwalter 0.13.2

- bugfix: Fix link to alternate frontend in default frontend
- feature: add `id` field to graphql status
- bugfix: fix server list display in api frontend

6.4 Verwalter 0.13.1

- feature: add `/v1/graphql` endpoint with GraphQL API
- feature: add `/v1/graphiql` for poking with GraphQL API
- feature: default frontend now shows peers having errors
- feature: default frontend now shows full list of failing roles with the links to logs under the navigation bar

6.5 Verwalter 0.13.0

- breaking: all requests to `/action` and `/wait_action` now require `Content-Type: application/json`
- feature: add support for `query.wasm` which might be used for overriding rendered roles and for custom queries
- feature: you can fetch current scheduler (and query) via API `/v1/wasm/scheduler.wasm` (only wasm scheduler though)

6.6 Verwalter 0.12.1

- Feature: add "node" variable to templates by default (in compatibility mode)

6.7 Verwalter 0.12.0

- We're preparing for list of roles and their variables be prepared by the wasm code in scheduler. This release only changes internals, preparing for that (we bump version to make a signal that things should be tested carefully).

6.8 Verwalter 0.11.3

- Feature: Added exerimental route `/v1/leader-redirect-by-node-name/` that returns redirect to a leader node
- Feature: Add a link to default frontend "common" frontend
- Bugfix: UI for 'Choice' variable type now works in api frontend

6.9 Verwalter 0.11.2

- bugfix: reset failures for the roles have been removed
- Feature: Add `CleanFiles` command

6.10 Verwalter 0.11.1

- feature: add `Condition` action which allows to execute an action if some files have been changed during executing other commands

6.11 Verwalter 0.11.0

- breaking: wasm scheduler requires returning object instead of tuple
- feature: new `SplitText` action, to deal with multiple generated files easily
- bugfix: wasm module will be reinitialized after panic

- bugfix: since verwalter 0.1.4 verwalter couldn't work as a single node
- breaking: serves `/files/` directory from static files

6.12 Verwalter 0.10.4

- feature: add an *experimental* `--allow-minority-cluster` option that allows verwalter to elect itself as a leader even if it sees less then N/2+1 nodes. I.e. in split-brain scenario two leaders might exist simultaneously which will then be merged. Note: this is a task of a specific scheduler to merge schedules appropriately.
- bugfix: additional css,js,fonts for alternative frontends were not served properly
- feature: allow to `--default-frontend` via CLI

6.13 Verwalter 0.10.3

- bugfix: timestamps in peer info now serialize as milliseconds since epoch
- wasm: add function to log panics
- wasm: add log/pow/exp functions needed for rust (actually llvm) build

6.14 Verwalter 0.10.2

- feature: upgrading trimmer to 0.3.6 allows to use escaping, dict and list literals in (.trm) templates
- Using `wasmi` instead of `parity-wasm` for interpreting wasm
- Initial routing for alternative frontends (`/~frontend-name/...` urls)

6.15 Verwalter 0.10.1

- Timeout for incoming requests changed 10sec -> 2 min (mostly important to download larger logs)
- Template variables are passed to renderer using temporary file rather than command-line (working around limitations of sudo command line)

6.16 Verwalter 0.10.0

- Experimental webassembly scheduler support

6.17 Verwalter 0.9.14

- UI: fix chunk size in log tailer, mistakenly committed debugging version
- scheduler: if scheduler continue to fail for 5 min verwalter restarts on this node (this effectively elects a new leader)

6.18 Verwalter 0.9.13

- UI: add “Skip to End” button on log tail, skip by default on pressing “follow”

6.19 Verwalter 0.9.12

- Bugfix: fix crash on serving empty log
- Bugfix: JS error on the last step of api-frontend pipeline
- Log viewer leads to tail with correct offset

6.20 Verwalter 0.9.11

- Bugfix: Content-Range headers on logs were invalid
- Api-frontend: sorted server list
- Api-frontend: no “delete daemon” when update is active

6.21 Verwalter 0.9.10

- Add nicer log tailing UI and activate link in role log list
- Add some cantal metrics
- Bugfix: list of peers did not display correct timestamps

6.22 Verwalter 0.9.9

- Bugfix: external logs were not served properly
- Bugfix: when cantal fails for some time, verwalter could block

6.23 Verwalter 0.9.8

- Keeps few backups of old schedules
- Updates dependencies of frontend

6.24 Verwalter 0.9.7

- Bugfix: when request to cantal failed, verwalter would never reconnect

6.25 Verwalter 0.9.6

- Settings tweak: runtime load watchdog timeout is increased to 5 sec
- Bugfix: fix "rerender all roles" button (broken in 0.9.0)

6.26 Verwalter 0.9.5

- Bugfix: because we used unbuffered reading of runtime, it was too slow, effectively preventing scheduler to start on larger schedules
- Settings tweak: scheduler watchdog timeout is increased to 5 sec

6.27 Verwalter 0.9.4

- Bugfix: follower was unable to render templates (only leader)

6.28 Verwalter 0.9.3

- Peer info (known since, last ping) is now visible again (broken in 0.9.0)

6.29 Verwalter 0.9.2

- Fix bug in showing old schedule at `/api/v1/schedule` api
- Logs now served by newer library, so bigger subset of requests supported (last modified, no range, ...)

6.30 Verwalter 0.9.1

- Release packaging fixes and few dependencies upgraded

6.31 Verwalter 0.9.0

The mayor change in this version of scheduler that we migrated from rotor network stack to tokio network stack. This is technically changes nothing from user point of view. But we also decided to drop/fix rarely used functions to make release more quick:

1. Dropped `/api/v1/scheduler` API, most useful info is now in `/api/v1/status` API
2. Some keys in status are changed
3. No metrics support any more, we'll reveal them in subsequent releases (we need more performant API in cantal for that)

Yes, we still use `/v1` and don't guarantee backwards compatibility between 0.x releases. That would be a major pain.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Index