Cantal Documentation

Release 0.6.10

Paul Colomiets

Aug 31, 2018

Contents

Contents:

CHAPTER 1

Concepts

1.1 Overview

The Cantal is a monitoring system designed specifically for real-time measuring and load-balancing distributed computing clusters.

The essential part of cantal is a **Cantal Agent** it does the following things:

- Scans for local metrics at **2 second interval**
- Preserves **one hour** history of all metrics data with *100% precision* (compressed)
- Provides **web interface** for viewing local metrics
- Has peer to peer **discovery** mechanism
- On demand provides aggregated statistics **over cluster**

Additional features:

- Cantal is aware of linux containers
- Almost zero-cost connunication between processes and agent
- Written in rust, so can track thousands of metrics with 2 second precision in less than couple of percents of a single CPU core

The project also consists of:

- Protocol to submit data to agent at nearly zero cost
- Command-line tool to view data locally without running agent

1.2 Background

Since Cantal is designed for real-time load balancing, it has very strong and very specific requirements:

1. Very high precision (cantal scans at 2 second interval, while common case is about a minute interval, very rarely interval is 10 seconds or less)
2. Similarly very fast collection of metrics across large cluster
3. Discovering trends quickly (i.e. having 30 value snapshots per minute we can find out load growth in a fraction of minute)
4. High availability (no master, quorum or similar)
5. Being able to observe individual nodes in case of partitioning
6. Lightweight

1.3 Design Decisions

Here is short roundup of all the important design decisions. Some of them are described in detail in the following sections.

Agent has embedded web server. So you can point your browser to:

```
http://node.domain.in.local.network:22682
```

..and see all the statistics on the node.

Agent stores history locally. So we don't loose stats in case of network failure

Agent has peer to peer gossip-like discovery with UDP. So we don't rely on any other discovery mechanism when time comes to gather metrics over cluster. Note: we do use UDP only for discovery, so we don't loose statistics when network is lossy.

You can ask **any instance** of agent to get metrics for whole cluster. This is how we allow to get data over whole cluster with a single HTTP request. But we do it *lazily*, so that we don't have full mesh of connections. I.e. when first client asks, we connect to every node by TCP and subcribe for metrics.

1.4 Discovery

We have gossip-like peer to peer discovery. You need to add a peer address at any node and every node will know it.

We use UDP for peer discovery. It works by sending ping packets between nodes. Each packet contains some critical info about current node and info about few randomly selected known peers.

Each node sends 5 ping packets with 10 neighbour nodes each second. Each ping packet receives pong packet with other 10 nodes. Overall it's not very large number of packets, and packets are distributed uniformly across the nodes. This allows to discover even large network with thousand of nodes in few dozens of seconds.

In the future we plan to discover physical topology using UDP packets. In turn this allows to display graph and provide diagnostics for different kinds of network partitions (including assymmetric partitions, bridge nodes, etc.)

Note that we use UDP exclusively for peer discovery. This allows us to *avoid* having a *full mesh* of TCP connections. But we don't use UDP for transferring metrics, so we don't lose statistics when network suddenly becomes lossy. Not being able to reach some nodes via UDP in lossy network is definitely the expected outcome and will help diagnose problems too.

1.5 Aggregated Metrics

To have efficient cluster management we consider imporant two points:

1. Resource manager don't have to gather from each node, they must be *pushed* as fast as possible
2. We don't want to constrain failover and/or lower the availability of the resource manager. I.e. cantal's data should be virtually always available for resource manager.

When we talk about *resource manager* (RM) we talk about any software which consumes metrics and implements some resource allocation decisions. Obviously resource management is out of scope of the cantal itself.

So to get metrics RM connects to any cantal agent and requests statistics for all it's peers.

On first request for some monitoring data Cantal Agent does the following:

1. Enables remote peer publish-subscribe subsystem
2. Connects to every known peer via bidirectional protocol (WebSockets in the current implementation)
3. Subscribes on each node for the subset of data requested by client
4. Fetches chunk of history for these metrics from every node

---

Note: UDP-based peer tracking and dicovery works always. So every agent does know all it's peers. We just activate TCP-based reliable publish-subscribe for metrics at request.

Other client might ask another node and that node will seamlessly provide the stats and historical data.

The only practical limitation of it is the that running a full-mesh of TCP connections is quite inefficient. So you should poll a single node while it's still available and switch to another one only when it's not.

Warning: It's hard to overstate that you should not poll every node in turn, otherwise you will have a full mesh of connections and every node will send updates to each other every two seconds.

Viewing web interface for **local metrics** and polling for them is perfectly OK on any and every node.

In perfect world we expect that resource manager will poll an agent on localhost, and has failover resource manager node with own cantal agent.

CHAPTER 2

Installation

2.1 Ubuntu

You need rust compiler:

```
wget https://static.rust-lang.org/dist/rust-1.1.0-x86_64-unknown-linux-gnu.tar.gz
tar -xzf rust-1.1.0-x86_64-unknown-linux-gnu.tar.gz
cd rust-1.1.0-x86_64-unknown-linux-gnu
sudo ./install.sh
```

Or you may use instructions on official website

Additional dependencies:

    sudo apt-get install build-essential libssl-dev

Then just download and build project with cargo:

```
wget https://github.com/tailhook/cantal/archive/staging.tar.gz
cd cantal-staging
cargo build --release
```

Note: We build from *staging* branch, because that contains javascripts already built. Building javascripts is a little bit more complex process, you shouldn't do, unless you're developing cantal itself.

Then you may either install it with:

```
make install
```

Optionally `DESTDIR` and `PREFIX` environment vars work.

Or you can build a package with:

```
checkinstall --default \
  --pkglicense=MIT --pkgname=cantal \
  --pkgversion="$(cat version.txt)" \
  --requires="libssl1.0.0"
  --nodoc --strip=no \
  make install
```

Additionally you need an `upstart` or `systemd` script to start cantal as a service. Here is one example:

```
start on runlevel [2345]
respawn
exec /usr/bin/cantal-agent --host 0.0.0.0 --port 22682 \
    --storage-dir /var/lib/cantal
```

CHAPTER 3

Configuration

3.1 Daemon Configuration

We're doing our best to keep cantal working without any configuration. But for achieving complex tast we need some configuration.

Important command-line options:

1. Enable cluster setup `--cluster-name=your-name`. Name must be the same on all nodes in the cluster (i.e. all nodes which should see each other)
2. Keep some metrics for restart `--storage-dir=/var/lib/cantal`. In clustered setup this also stores list of peers, so that if all the nodes are restarted simultaneously, they discover each other

3.1.1 Cluster Setup

Another piece of cluster setup is: introduce nodes to each other:

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

This only works if `cluster-name` matches and after nodes are able to interchange ping-pong packets between each other (also `machine-id` must be different which is usually provided by the system).

3.2 Carbon Integration

Carbon integration allows to use cantal as an agent for carbon, so you can view the data in graphite or any other carbon-compatible system (such as a graphana)

Basically this allows you to view recent data in cantal and use carbon for archival of statistics

Note: The support is currently far from be comprehensive. Only some data can be sent to carbon. Sending whole collected statistics to graphite is too much, so we adding features one by one.

3.2.1 Configuration

Cantal starting with v0.3.0, has a default configuration directory `/etc/cantal`. You need to put some configuration file there:

```
# /etc/cantal/localhost.carbon.yaml
host: localhost
port: 2003
interval: 10
enable-cgroup-stats: true
enable-application-metrics: true
```

All configurations which end with `.carbon.yaml` will be read. Multiple configurations may be used, each configuration is a separate connection with it's own set of metrics.

Options:

host (required) The **IP address** to send data to. *Hostnames are not supported yet*.

port (default `2003`) Port where carbon listens with text protocol. The default matches the same of carbon.

interval (default `10`) Interval of sending data to carbon. The cantal's collection interval is `2` seconds for most metrics. But there is no much value of sending such detailed statistics to carbon. Cantal will provide 1 hour of highest precision history in it's own interface and send averages of the values to a carbon.

enable-cgroup-stats (default `false`) Send data about cgroups to carbon

enable-application-metrics (default `false`) Send data with application metrics to carbon. The application must have an **unique** `CANTAL_APPNAME` in environment to have metrics delivered to carbon. Anyway `CANTAL_APPNAME` is ignored if application is in cgroup.

3.2.2 Metrics Layout

By default cantal sends nothing, even if connection params are set.

CGroup statistics (enabled with `enable-cgroup-stats`):

- `cantal.<CLUSTER_NAME>.<HOSTNAME>.cgroups.<GROUP_NAME>.<METRIC_NAME>`
  - Metrics (all represent the sum for all processes in the group):
    - `vsize` – virtual memory size
    - `rss` – resident set size
    - `num_processes` – total number of processes in the group
    - `num_threads` – total number of threads in the group
    - `user_cpu_percent` – percentage of CPU spent in user mode
    - `system_cpu_percent` – percentage of CPU spent in system mode
    - `read_bps` – average bytes per second read on disk
    - `writes_bps` – average bytes per second written to disk

  - Ggroup is a dot-delimited hierarchy of cgroups with systemd-like suffixes removed, for example: `/sys/fs/cgroup/systemd/system.slice/nscd.service` will turn into `system.nscd`
  - The `.swap` and `.mount` (systemd-specific) groups are skipped
  - The root group `user` (upstart- and systemd-specific) group is ignored
  - If the process is in group `a.b` it will not count for group `a`, the statistics for `a` contains only processes immediately in the group

- `cantal.<CLUSTER_NAME>.<HOSTNAME>.cgroups.<GROUP_NAME>.states.<STATE_NAME>.<METRIC_NAME>` – application-submitted metrics which have a `state` value
- `cantal.<CLUSTER_NAME>.<HOSTNAME>.cgroups.<GROUP_NAME>.groups.<STATE_NAME>.<METRIC_NAME>` – application-submitted metrics which have a `group` value

Application metrics that are outside of cgroups have similar layout but do not have any system metrics yet (enabled with `enable-application-metrics`):

- `cantal.<CLUSTER_NAME>.<HOSTNAME>.apps.<APPLICATION_NAME>.states.<STATE_NAME>.<METRIC_NAME>` – application-submitted metrics which have a `state` value
- `cantal.<CLUSTER_NAME>.<HOSTNAME>.apps.<APPLICATION_NAME>.groups.<STATE_NAME>.<METRIC_NAME>` – application-submitted metrics which have a `group` value

`CLUSTER_NAME` is `no-cluster` if no `--cluster-name=something` is specified in the command-line.

`APPLICATION_NAME` is the value of `CANTAL_APPNAME` environment variable that exists alongside with the `CANTAL_PATH`.

CHAPTER 4

API

4.1 Policy

Currently Cantal has `/v1/` API. We don't increment API version on backwards compatible changes. The following is deemed backwards-compatible:

- Addition of new resources
- Addition of new fields in structures
- Deprecation (but not removal) of resources
- Deprecation (but not removal) of fields in structures
- New formats of output (with some negotiation way, i.e. Accept header)

The (backwards-compatible) changes in API are listed here by version of a Cantal agent itself.

Until cantal reaches `1.0` it's only guaranteed to support single API version, after `1.0` we will support previous version of API for several releases after new API is introduced.

CHAPTER 5

Memory Map Protocol

5.1 Motivation

Cantal scans the whole system at 2 second interval. This includes metrics of your application. If cantal would poll applications by some kind of remote procedure call (RPC), it would rely too much on the application responsiveness to provide fine-grained statistics. For many synchronous applications it just doesn't work, because it may wait more than 2 seconds for a database on occassion).

Cantal is implemented with another kind of inter-process communication (IPC), the shared memory. As you will see later in the text it requires almost zero configuration and allows efficient collection of statistics for most kinds of programs. For scripting languages it's also practially zero-cost. For fully-threaded programs it's usually as cheap as any other way you could implement.

For scripting languages using shared memory appoach described here also allows to dive into the application that is currently slow or unresponsive.

5.2 Overview of Files and Discovery

The metrics are discovered by cantal by scanning environment variables of running processes. Whenever it sees `CANTAL_PATH` in environment the metrics are gathered from there.

For `CANTAL_PATH=/run/myapp`, catal will look into:

- `/run/myapp.meta` for metadata (names size and alignemnt) for metrics
- `/run/myapp.values` for metrics

Here is a short example of the contents of the meta file:

```
counter 8: {"metric": "requests.number"}
counter 8: {"metric": "requests.duration", "unit": "ms"}
```

It contains two 8 byte (64bit) unsigned integers, which are growing counters. Here is the respective `.values` file (displayed in a format of `hexdump -C`):

```
00000000  61 00 00 00 00 00 00 00  67 62 00 00 00 00 00 00  |a.......gb......|
```

Here we can see that there have been 97 requests (0x61) each lasts of almost 260 milleconds on average (0x6261/0x61).

The files must reside on some in-memory file system (`tmpfs`). On typical system `/run` folder is a good place. On some systems `/tmp` is a `tmpfs` too, but be careful not to put it into HDD or SSD. In docker containers you need to map some tmpfs folder from host system (example command line: `docker run -v /run/containers/my1:/run/cantal -e CANTAL_PATH=/run/cantal ...`) In container running by lithos a `!Statedir` is a good place.

5.3 Metadata File Format

File format of the meta data is simple: every next line is the metric (or a padding). Format of the line:

```
TYPE NUMBER_OF_BYTES TYPE_PARAM: JSON_METADATA
```

For example:

```
level 8 signed:  {"metric": "memoryusage"}
counter 8:  {"metric": "requests_processed"}
```

The `TYPE_PARAM` is optional and is currently used for `level` type, which can be one of the `signed` or `float` (`unsigned` will be added in future)

The `JSON_METADATA` field is a subset of a JSON, and is currently limited to (we may extend it to a larger subset of or full JSON later):

1. Serialized data should contain no newlines (you can't pretty print json)
2. Only a dictionary (object) with string keys and string values is supported

The keys and the values of the dictionary might be arbitrary. But the whole set of keys must be unique for the file.

An a padding is just:

```
pad 123
```

Where `123` is the number of bytes.

The values of the respective lengths are stored consecutively in the `.values` file in the same order as entries in metadata. The `pad` entries might be used to align counters to addresses of multiples of 8, or whatever is needed for efficient accounting.

Metadata file is **imutable**. To create a metadata file you must write to a temporary name then do an atomic rename operation to put it to the right path.

5.4 Values File Format

Values file is a binary file that contains raw values in host byte order written consecutively one after another with actually any file structure, or in other words with the structure defined in metadata file. For example, if we have a metadata of:

```
counter 8: {"metric": "requests.number"}
counter 8: {"metric": "requests.duration", "unit": "ms"}
pad 48
state 64: {"value": "request.sql.request"}
```

There are exactly:

- 8 bytes, integer in host byte order, counter of the number of requests
- 8 bytes, integer in host byte order, sum of the duration of all requests
- 48 bytes of padding, any garbage can be there, but usually just zeros
- 64 bytes state, first bytes of the sql request that is currently going on

The file size is 128 bytes. As you can see the state is aligned to 64 bytes, because this makes it another CPU cache line. This means two processors can write counters and state simultatenously without any kind of contention. This is very CPU-dependent and optional for file format, but usually some kind of padding is implemented by the implementation.

Overall, the structure of the file is implemented this way so that program can atomically adjust any counter directly in shared memory without ever duplicating metrics or delaying the statistics submission (for some very fast and heavy-threaded programs it may still be a lot of contention and traditional technics may be applied here, but please do benchmarks first).

5.5 Data Types

Data types that are currently supported by cantal agent:

Type Name	Allowed Sizes	Alignment (recommended)	Description
`counter`	8 bytes (64bit)	8 bytes	A 64bit ever-growing counter.
`level`	8 bytes (64bit)	8 bytes	A current value of something, may grow or decrease
`state`	16-65535 bytes	64 bytes	An arbitrary string value that is visible in cantal. No history of it is stored.
`pad`	1-65535 bytes	–	No data

More types and sizes will be implemented later.

The `counter` value is a most useful type. You should increment the value of counter using atomic operations (unless you have a GIL so any small write is atomic) and never write whole value to it. It's fine to initialize counter value to zero on application restart, you don't need to store value somewhere.

Good use cases for `counter` are:

1. Number of requests
2. Total duration of requests
3. Tasks processed

From the above you can derive the following values, which you **should not write by the application**, but they are calculated by a cantal itself:

1. The number requests per second (or any other unit in time)

2. The average duration of each request
3. Tasks processed per second

Good use cases for `level` are:

1. Memory used by object pool
2. Current queue size

Don't use `level` *for things that are number of operations per second or similar things. Use* `counter` *instead. This allows correct statistics even if collection interval changes, when something is slow and so on.*

CHAPTER 6

Cantal Changes by Version

6.1 Cantal 0.6.10

- Bugfix: previously swap metric wasn't seen in metrics (only in all_processes) and 0 was always submitted to graphite/carbon

6.2 Cantal 0.6.9

- Bugfix: previously cantal sometimes skipped some processes because scanning /proc is not atomic. Now we decrease the issue by making scan fast and doing it twice.

6.3 Cantal 0.6.8

- Bugfix: machine uptime and process' uptimes were broken in UI

6.4 Cantal 0.6.7

- Bugfix: previously cantal could crash when time jumps backwards. Currently, it waits (in metrics scanner code) if delta is < 10 sec and crashes with clear log message for large time jumps.

6.5 Cantal 0.6.6

- Bugfix: when a `CANTAL_PATH` referring to a same file is encountered in multiple processes we no longer duplicate metrics

6.6 Cantal 0.6.5

- Feature: add `peers` graphql field

6.7 Cantal 0.6.4

- Feature: add `local.cgroups` graphql endpoint
- Feature: add `local.processes` graphql endpoint

6.8 Cantal 0.6.3

- Bugfix: add `num_peers`, `num_stale` back to `/status.json`, same fields added to graphql endpoint

6.9 Cantal 0.6.2

- Bugfix: larger timeouts for incoming http requests
- Bugfix: add `version` back to `/status.json`

6.10 Cantal 0.6.1

- Bugfix: fix JS error on /local/peers page

6.11 Cantal 0.6.0

- We reworked network subsystem to use tokio instead of home-grown async, this looses some features for now, but is an important step for future
- Breaking: remote subsystem doesn't work, including the whole `/remote` route, we will be working to add feature back soon
- Feature: add graphql API (only status for now)
- Breaking: `/status.json` contains less data, use graphql API

CHAPTER 7

Indices and tables

- genindex
- modindex
- search