# Leela Documentation

*Release 3.2.0*

**Diego Souza**

January 29, 2016

# Contents

Leela is a system that allows you to store, retrieve and monitor the performance metrics of your systems in real-time using a variety of protocols.

For data collection, currently you can use collectd or a fairly simple text protocol over UDP. This gives you a decent coverage of standard systems and applications at the same making it very easy to collect custom metrics.

Data retrieval can be done using a restful API which makes it easier to create dashboards, analyze historical data or simply plot graphs in the browser.

You can even monitor real-time data using XMPP protocol. Simply register a query and you will start receiving events in JSON as soon as they are received by the server.

CHAPTER 1

# General

CHAPTER 2

# Administrators

- `Configuring leela`
- `Configuring cassandra`
- `Configuring ejabberd`
- `Configuring redis`
- `Monitoring and tuning`

# CHAPTER 3

# Users

## 3.1 Writing

- `Collectd interface`
- UDP interface

## 3.2 Querying

- REST API
- `Dashboard`

## 3.3 Monitoring

- XMPP interface
- DMPROC protocol

## 3.4 Developers

- `Javascript library`
- `Python library`
- `Ruby library`
- `Haskell library`

CHAPTER 4

# Changelog

- https://github.com/locaweb/leela/blob/master/CHANGELOG

# CHAPTER 5

# License

APACHE 2.0

CHAPTER 6

# Author

- dgvncsz0f <dsouza@c0d3.xxx>

CHAPTER 7

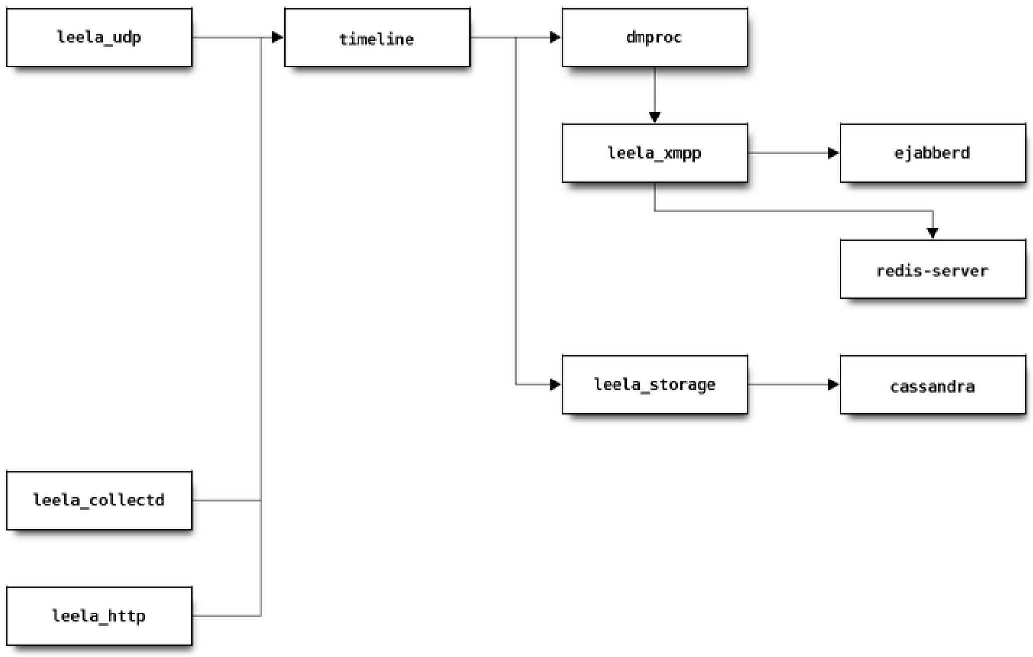
# Contributors

- Juliano Martinez [former Author (v0.0.9)]
- Rodrigo Sampaio Vaz

## 7.1 Leela Architecture

Leela is designed to run on Linux. Although I believe it should works on other POSIX platforms, the only environment we have tested it is Linux.

It makes heavy use of unix sockets (mostly datagrams) and is currently written in python and haskell. The following diagram summarizes the major components:



In the above diagram all components communicate using unix sockets, but external systems [cassandra, redis-server and ejabberd] which use TCP. This [the use of unix sockets] implies that everything must run on the same machine, or that you will a machine with a reasonable number of cores to sustain high loads.

Following we provide more details about each component and how they interact with each other.

### 7.1.1 Leela UDP/Collectd

They simply parse the packet and forward to the timeline using the leela internal protocol.

### 7.1.2 Leela HTTP

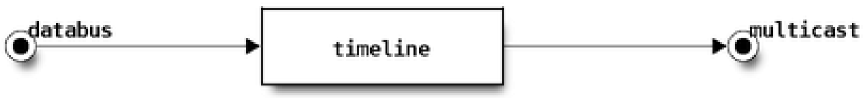
This exposes the rest API which allows you to retrieve historical data. It reads data from cassandra directly but in the future all reading and writing to the storage will be go through the leela-storage service.

The HTTP provides a read/write interface. Writing are simply forward to the timeline, as the previous components do.

### 7.1.3 Timeline

This is the only component that effectively knows about metrics. You may think of it as a function that takes a *metric* and produces one or more *events* [an event is just a gauge type].

The process exposes the following unix sockets:

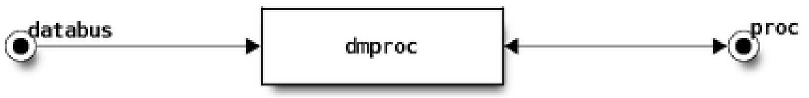


The databus socket is the one metrics should arrive. Each frontend [udp, collectd, http] writes one or more metrics into it. Then, if any given metric generates any event then the timeline writes them into the connected peers. This is done using the *multicast* unix socket.

Interested processes should continuously register themselves using the *multicast* socket in order to receive the events that the timeline generates.

The reason this must be a continuous operation is that the timeline purges dead nodes, i.e., nodes that are not sending register messages in a timely manner.

### 7.1.4 DMPROC



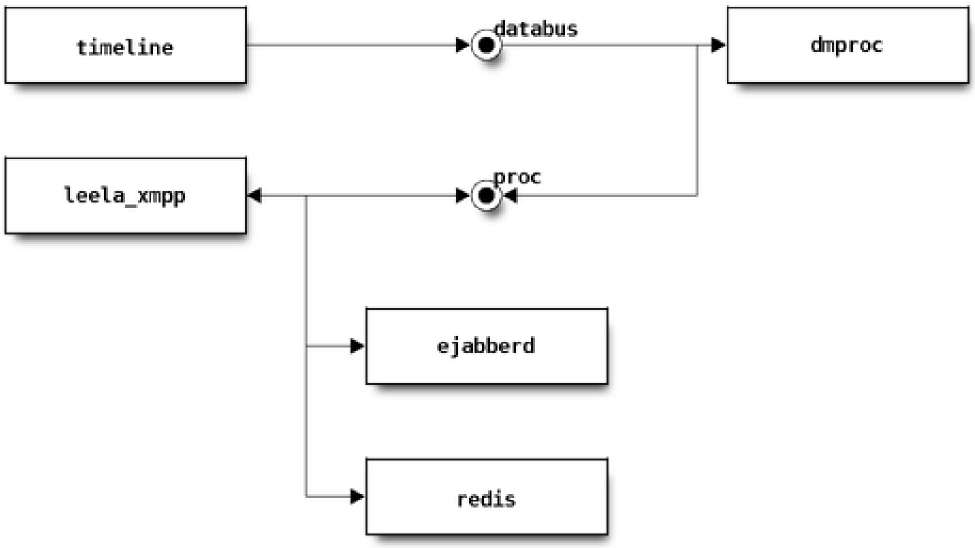
This is the engine that allows users to monitor metrics as soon as they are received. Once started, it register itself in timeline in order to receive events into the *databus* socket, and exposes its service through the *proc* socket.

The proc socket is the only one that is stream oriented. The protocol is fairly simple though. It prefixes all packets with its size, using a unsigned short [2 bytes] big endian encoded.

```
0                      2
|          |           |
-----------+-----------+
|         size         |
+----------------------+
|                      |
|        payload       |
|        (0-65k)        |
|                       |
+-----------------------+
```

### 7.1.5 Leela XMPP

Exposes its services as an user of a XMPP service using a language that resembles SQL. This module allows one to monitor metrics in real time via XMPP.



The redis is used as a directory service. When a request is made by a user an new entry is written into the redis. Periodically, leela xmpp service reads from redis in order to know which users are requesting information. When a new entry is found, it establishes a connection with dmproc and any output is forwarded via XMPP to the users requesting the information. Similarly, whenever an entry is removed from redis the associated connection with dmproc is closed.

The load on a redis server is very low, but it is extremely important to make sure it is always available. If the redis service become unavailable, so does the leela-xmpp.

## 7.2 Installing the Debian Package

Currently we don't provide any binary packages, so you must build it on your own. Although the package is suitable for using with debian stables [= squeeze at the time we wrote this] building it depends on packages only found on experimental branch: `ghc` and `cabal-install`.

If you can workaround these, building it should be fairly straightforward:

```
$ git clone git://github.com/locaweb/leela.git
$ cd leela
$ debuild
```

After that you can install it using `dpkg`:

```
$ sudo dpkg -i ../leela_<version>_<arch>.deb
$ sudo apt-get -f install
```

## 7.3 Installing the Linux Tarball

This method uses virtualenv to install leela.

### 7.3.1 Requirements

- python2
- virtualenv
- ghc7
- cabal

On arch linux, the above can be fulfilled using pacman:

```
$ pacman -S python2-virtualenv cabal-install
```

Now, download the latest release from github and build it:

```
$ wget -Oleela-v2x.tar.gz https://github.com/locaweb/leela/archive/v2.x.tar.gz
$ tar -xf leela-v2x.tar.gz
$ cd leela-2.x
$ make bootstrap
$ make dist-build
```

And install it:

```
$ root=/opt/leela
$ make dist-install root=$root
```

You should check the configuration file [= *$root/etc/leela.conf*] and make sure everything checks out. After that, start the service.

```
$ $root/etc/init.d/leela start
```

## 7.4 dmproc

This is the stream processor engine, which is a fancy name to a component that is able to apply functions over a set of events.

It is important to notice that you do not interact with this directly. Rather it is used by other components, like `xmpp`, to fulfill a user request.

Nevertheless, as the functions available here are usually used *as-is* by users through these *higher level* components, the first part of this documentation focuses on the available functions, whereas the second part contains detailed explanation about the protocol.

### 7.4.1 Function syntax

Each function contains a name, usually as ascii encoded string, followed by zero or more arguments. For instance, the `maximum` function, which computes the maximum value of a set of events, takes no arguments. Operators are the sole exception to this rule, as they are enclosed by parenthesis or brackets. Currently all functions (as well as operators) depends only on the input. They will only produce a result when it receives one or more events. In other words, they are pure functions.

You may compose functions using the `|` (pipe) operator which resembles greatly the unix pipe operator. So you may think of each function as a combinator and using the pipe operator allow you to grow bigger pipelines. For instance, suppose you have a event that ranges from *0* to *1*, the following function should compute the *mean* of the current set of events and then multiply this value by *100* (left to right application), only returning those *greater than 50*:

```
mean | (* 100) | [> 50]
```

Operators are always enclosed by parenthesis or brackets. Parenthesis are used for arithmetic operators whereas brackets are used for comparison operators. The syntax allow the operator to be placed either on the left or right side. For instance, `(/ 100)` will divide by 100 whereas `(100 /)` will divide 100 by the value of the event. The same applies for comparison operators.

There are four arithmetic operators defined: `*`, `+`, `-` and `/`, which respectively computes the *multiplication*, *addition*, *subtraction* and *division*.

And there are six comparison operators defined: > [greater than], < [less than], >= [greater than or equal to], <= [less than or equal to], = [equal to] and /= [not equal to].

## 7.4.2 Functions reference

### abs

Computes the absolute value a number.

### ceil

Smallest integral value that is not less than the current number.

### floor

Largest integral value not greater than the current value.

### round

Round towards infinity.

### truncate

Round towards zero.

### mean

The *mean* of the set of events.

### sma :n

Computes the simple moving average. The actual implementation makes use of `ewma` using `1-2/(n+1)` as the `alpha` parameter.

### ewma :alpha

Computes the exponential weighted moving average using :alpha as the *alpha* parameter.

**Arguments:**

**n** A positive integer number;

### sample :n/:m

Samples n elements out from a population of m items. The exact frequency of elements generated by this function is defined as follows. Let L the total of elements:

```
n * (L (mod m)) + minimum n (L (mod m))
```

**Arguments:**

**n** The number of elements pick out of *m*;

**m** The so called population size;

**Example:**:

```
sma 30 | sample 1/5
```

### minimum

The minimum value.

### maximum

The maximum value.

### prod

Multiplication of all values in the stream.

### sum

Summation of all values in the stream.

### window :n :pipeline

Creates an window of n items and when this is full applies a function to produce a result.

**Arguments:**

**n** An positive integer number [ n > 0 ], which defines the size of the window;

**pipeline** The function to apply, enclosed with parenthesis. You may use the pipe to combine multiple functions, and all functions here defined but sma, sample and window;

**Example:**:

```
window 30 (mean | (* 100))
```

### id

The identity function: id x == x.

### 7.4.3 Operators reference

#### Arithmetic

**+** Addition (e.g.: `(+ n)` or `(n +)`);

**-** Subtraction (e.g.: `(- n)` or `(n -)`);

***** Multiplication (e.g.: `(* n)` or `(n *)`);

**/** Division (e.g.: `(/ n)` or `(n /)`);

#### Comparison

**>** Greater than (e.g: `[> n]` or `[n >]`)

**>=** Greater than or equal to (e.g: `[>= n]` or `[n >=]`)

**<=** Less than equal to (e.g: `[<= n]` or `[n <=]`)

**>** Less than (e.g: `[< n]` or `[< n]`)

**=** Equal to (e.g: `[n =]` or `[= n]`)

**/=** Not equal to (e.g: `[n /=]` or `[/= n]`)

## 7.5 REST API

This exposes datas via a *REST* interface. The following should apply to all resources:

- All resources support the *JSON-P* protocol by appending the `callback` parameter to the URL:

```
/v1/foobar?callback=my_handler
```

- You may add `debug=true` to enable debugging information. This can give you a hint of what went wrong:

```
/v1/foobar?debug=true
```

### 7.5.1 Common Response Codes

**2xx** Ok;

**4xx** Client error;

**5xx** Server error;

**200** Success;

**201** Created;

**404** The requested data could not be found [invalid range, missing event etc.];

**400** You did something wrong;

**500** Internal server error;

### 7.5.2 Error Responses

They will always come using the following format:

```
{"status": int, "reason": string}
```

**status** the http response code [e.g. 404, 500];

**reason** a very short description of what went wrong [might not be that useful though, use `debug=true` for more context];

## 7.5.3 Resources

#### /v1/data/:year/:month/:key

#### /v1/:year/:month/:key

##### Method: `GET`

Retrieves all events/data withing a given month.

**status**

- 200 ok
- 404 not found
- 400 invalid range
- xxx error

**query string**

- nan=purge: Removes all *nan/infinty* from the response;
- nan=allow: The default, allow *nan/infinity* values to appear on the response;

#### /v1/data/:year/:month/:day/:key

#### /v1/:year/:month/:day/:key

##### Method: `GET`

Retrieves all events/data withing a given day.

**status**

- 200 ok
- 404 not found
- xxx error

**query string**

- nan=purge: Removes all *nan/infinty* from the response;
- nan=allow: The default, allow *nan/infinity* values to appear on the response;

#### /v1/data/past24/:key

#### /v1/past24/:key

##### Method: `GET`

Retrieves data/events from the past 24 hours.

**status**

- 200 ok

- 404 not found
- xxx error

**query string**

- nan=purge: Removes all *nan/infinty* from the response;
- nan=allow: The default, allow *nan/infinity* values to appear on the response;

### /v1/data/pastweek/:key

### /v1/pastweek/:key

#### Method: `GET`

Retrieves data/events from the past week.

**status**

- 200 ok
- 404 not found
- xxx error

**query string**

- nan=purge: Removes all *nan/infinty* from the response;
- nan=allow: The default, allow *nan/infinity* values to appear on the response;

### /v1/data/:key

### /v1/:key

#### Method: `GET`

Retrieves data/events within a given time range.

**status**

- 200 ok
- 404 not found
- 400 invalid range
- xxx error

**query string**

- nan=purge: Removes all *nan/infinty* from the response;
- nan=allow: The default, allow *nan/infinity* values to appear on the response;
- start=TIMESPEC: The start time [UTC]. Make sure `finish >= start`;
- finish=TIMESPEC: The finish data [UTC];

TIMESPEC uses the the following *strftime* format:

```
%Y%m%dT%H%M
```

Example:

```
$ curl {endpoint}/v1/foobar?start=20120101T1430&finish=20120101T1500
{ "status": 200,
  "results": ...
}
```

### /v1/:key

#### Method: `POST`

Inserts a new metric under this key. The body of the request must be a valid json and the json must have the following keys:

**status**

- 201 ok
- 400 bad/missing required values
- xxx error

**parameters**

- type: One of `gauge`, `counter`, `derive`, `absolute`
- name: [optional] The name to store this metric. If this is provided, it must match the one given on the path;
- value: The value to store under this key/timestamp;
- timestamp: [optional] Unix timestamp [number of seconds since epoch];

You may also provide a list of metrics as long as theirs names match the on given on the URL.

Examples:

```
$ curl -X POST -d '{"type": "gauge", "value": 0.2}' {endpoint}/v1/foobar
{"status": 201,
 "results": [{"name": "foobar", "timestamp": 1366549812, "type": "gauge", "value": 0.2}]
}
```

### /v1/data/:key

#### Method: `PUT`

*Deprecated: use /POST/*

#### Method: `POST`

Inserts a new data value under this key. The body of the request must be a valid json, and the json must have the following keys:

**status**

- 201 ok
- 400 bad/missing required values
- xxx error

**parameters**

- name: [optional] The name to store this object. This must match the name given on the URL;

- value: The value to store under this key/timestamp;
- timestamp: [optional] Unix timestamp [number of seconds since epoch];

You may use this resource to store up to 8k bytes worth of data [in the `value` field]. You may also provide a list of values [as long as theirs names match the one given on the URL] in which case each item of the list is subject to this limit.

Example:

```
$ curl -X POST -d '{"value": :VALUE, "timestamp": 1352483918}' {endpoint}/v1/data/foobar
{ "status": 201,
  "results": [{"name": "foobar", "timestamp": 1352483918, "value": :VALUE}]
}
```

## 7.6 UDP Interface

This is a write-only interface. It uses UDP and the protocol is plain text. The protocol is fairly simple:

```
<type> <length>|<key> <value>[ timestamp];
```

**type**

- gauge
- derive
- counter
- absolute

**length** The size of the *key* string;

**key** Any string (ascii encoded), up to 255 characters;

**value** Any double value (e.g.: 0.0, nan, 3.2e12);

**timestamp** [optional] the unix timestamp you want to store this event. If you don't provide this value the server will use the current timestamp;

There is no *ack* [=confirmation the event was received], nor authentication, nor checksum [application level] whatsoever. If you need such a feature, use a different protocol [e.g. collectd].

N.B.: There is no trailing newline here. Adding a trailing newline is a parser error.

### 7.6.1 PING

The UDP protocol is also capable of receiving a PING message that can use used to test connectivity. The syntax is as follows:

```
ping\n
```

### 7.6.2 Examples

Assuming you have *netcat*, and the server up and running, the following shell commands should work:

```
# the ping message
$ echo ping                                     | nc -u localhost 6968
pong

$ echo -n "gauge 10|example.e0 0.75 1350332001;" | nc -u localhost 6968
$ echo -n "derive 10|example.e0 0.76;"          | nc -u localhost 6968
```

### 7.6.3 Legacy udp protocol

This protocol is deprecated. It will be removed in future releases:

```
<name>: <value>[ timestamp]\n
```

## 7.7 XMPP Interface

The *XMPP* protocol allows you to monitor real time events. The protocol supports a simple query language that enables one to transform the events in a suitable manner.

The xmpp interface uses a simple *request-response* protocol and all commands are executed modulo de current user. In other words, the commands are isolated by the current user account.

Following a list of commands that are currently supported.

### 7.7.1 SELECT * FROM leela.xmpp;

Returns the current registered functions. Example:

```
SELECT * FROM leela.xmpp;
{ "status": 200,
  "results": [ { "cmd": "SELECT id FROM hm6177.cpu.cpu.idle;",
                 "key": "b0acdfe11875c074c760bfa8e34da49c1dfe73bd998bf720efc349c6bfd31d756d9b88f2
               }
             ]
}
```

The `results` entry contains an object with the following keys:

**cmd** The registered query;

**key** An opaque string that references this query. You may use this in a `DELETE` command to unregister this query;

### 7.7.2 SELECT :proc FROM :regex;

Registers a new function to monitor real time events. Example:

```
SELECT id FROM ^.*.cpu.cpu.idle$;
{ "status": 200,
  "results": { "key": "baa7163f7b51c3e96d7ee54e08a147840c1c2a682c89cbae2edd288506954dd568980394a8
             }
}
```

The `regex` is a posix regular expression and `proc` is a function to apply over the events that matches the regex. The complete reference may be found at `dmproc`.

This command returns an structure with the following keys:

**key** An opaque string that references this query. You may use this in a `DELETE` command to unregister this query;

Then for each event that is generated by the registered function the following message is created:

```
{ "status": 200, "results": { "event", { "name": "...",
                                         "timestamp": 1350334144.0,
                                         "value": 0.8553317028766958
                                       }
                            }
}
```

### 7.7.3 DELETE FROM leela.xmpp;

Unregister all functions registered for this account. Example:

```
DELETE FROM leela.xmpp;
{ "status": 200,
  "results": [ { "key": "baa7163f7b51c3e96d7ee54e08a147840c1c2a682c89cbae2edd288506954dd568980394
               }
             ]
}
```

#### DELETE FROM leela.xmpp WHERE key=:key;

Unregister a function referenced by a given key. Example:

```
DELETE FROM leela.xmpp WHERE key=284692849396a112668bbaa3dbc30e9d5c097c31998ec0569938d8cb0aaee9a2
{ "status": 200,
  "results": { "key": "284692849396a112668bbaa3dbc30e9d5c097c31998ec0569938d8cb0aaee9a282852fa56c
             }
}
```

### 7.7.4 Response structure

All messages follows this structure:

```
{"status":INTEGER, "debug":OBJECT, "reason":STRING, "results":OBJECT}
```

#### Status

**2xx** Ok;

**200** Success;

**201** Created;

**4xx** Client error;

**404** The requested data could not be found (invalid range, missing event etc.);

**400** You did something wrong;

**5xx** Server error;

**500** Internal server error;

**503** Maintanance;

#### Reason

In case of an error, this provides an human readable message to help you debug the root cause.

#### Results

The object you requested for. This vary greatly depending on the command.