Komlogd Documentation

Release 0.1.0

Komlog

February 13, 2017

Contents

Komlogd is the agent for communicating with Komlog platform, a distributed execution platform aimed at visualizing and sharing time series.

Komlogd is built in python and can be used as an independent daemon or as an API, so you can add Komlog functionality to your own python code.

Automatic install:

```
pip install komlogd
```

Komlogd is listed in PyPI so you can install with `pip` or `easy_install`.

Requirements

Komlogd needs python 3.5+. It also has some aditional dependencies (that will install automatically if you use `pip`):

- aiohttp
- cryptography
- pandas
- pyyaml

Note: To install dependencies, pip will need some of your distribution packages, like `gcc`, `libffi-dev`, `numpy`, etc. On slow devices, installing pandas and cryptography can take a lot of time, so maybe you prefer to install them directly using your distribution's package manager.

Docs

CHAPTER 1

Install and first steps

1.1 Automatic install

komlogd is on PyPI and can be installed with `pip` or `easy_install`:

```
pip install komlogd
```

1.2 Requirements

komlogd needs Python 3.5+. It also has some aditional requirements (that will install automatically with `pip`):

- aiohttp
- cryptography
- pandas
- pyyaml

Note: To install dependencies, pip will need some of your distribution packages, like `gcc`, `libffi-dev`, `numpy`, etc. On slow devices, installing pandas and cryptography can take a lot of time, so maybe you prefer to install them directly using your distribution's package manager.

1.3 First execution

Once installed, we can start komlogd with this command:

```
komlogd &
```

During its first execution, komlogd will create and initialize the necesary files and RSA keys needed for communicating with Komlog. After the initialization komlogd will terminate.

By default komlogd will create a new directory inside user's `$HOME` directory called `.komlogd` with the following contents:

```
.komlogd
-- key.priv
-- key.pub
```

```
-- komlogd.yaml
-- log
    -- komlogd.log
```

Contents created are:

- **key.priv**: private RSA key file.
- **key.pub**: public RSA key file. This key is the one we have to add at Komlog web to authorize the agent in Komlog.
- **komlogd.yaml**: main configuration file.
- **log/komlogd.log**: log file.

> **Note:** Private key will never be sent to Komlog.

At this moment, we can start with komlogd *Configuration*.

CHAPTER 2

Configuration

2.1 Komlog Access Configuration

Once komlogd file and directory structure is initialized, as explained at *First execution*, is time to proceed to komlogd configuration so it can communicate with Komlog.

2.1.1 User and RSA Key Configuration

To access Komlog we need to set username and RSA key in komlogd configuration file (**komlogd.yaml**).

The username is set with this key:

```
- username: <username>
```

Replacing <*username*> with our Komlog username.

komlogd RSA key is generated during the first execution, and placed in `$HOME/.komlogd/key.priv` by default. komlogd will use this key for the authentication process. However, if we want to store the key in another file, or use another key, we should set the following key:

```
- keyfile: <path_to_key.priv>
```

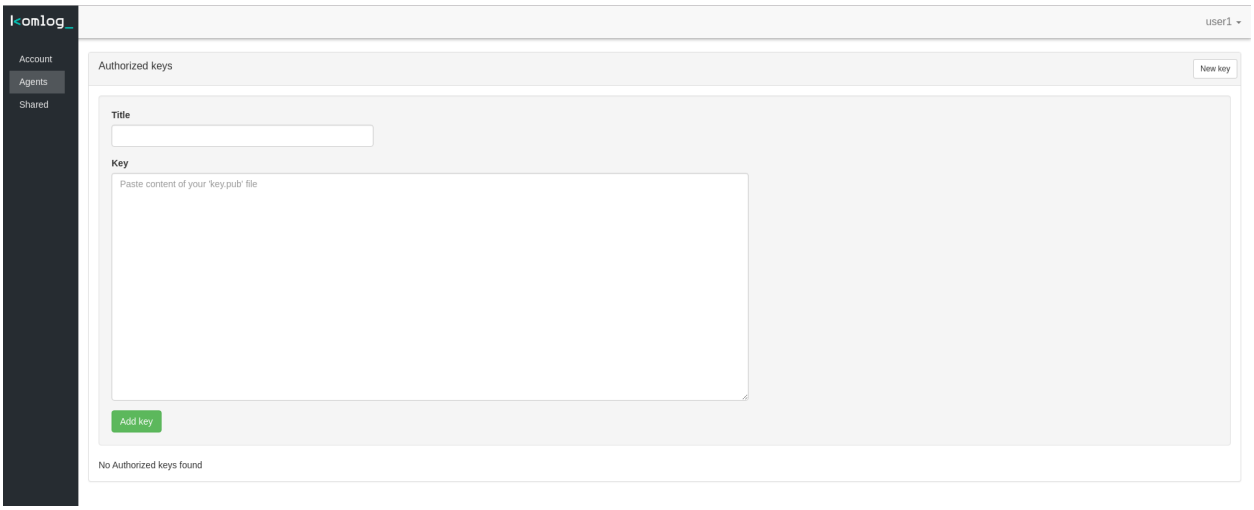
Replacing <*path_to_key.priv*> with the filename and full path of our key.

Note: komlogd private key will never be sent to Komlog.

2.1.2 komlogd authorization

You have to add komlogd's public key to the list of authorized keys so that komlogd can succeed in the authentication process. To do it, follow these steps:

- Access configuration page with your Komlog user.
- Go to `Agents` menu and click `New Agent` button.
- A form will be shown with two fields. The first one is to set the name of the agent, and the second one to paste the contents of its public key. You can find the public key in file `$HOME/.komlogd/key.pub`.



2.2 Scheduled jobs configuration

komlogd lets you schedule periodic jobs and send its outputs to Komlog. This functionality allows you to see commands or scripts outputs on Komlog web, identify variables directly on the texts, share them to other users, etc.

Supose we want to send to Komlog the result of executing the command **df -k** every hour.

To accomplish that, we should add to komlogd's configuration file (**komlogd.yaml**) the following **job** block:

```
- job:
    uri: system.info.disk
    command: df -k
    enabled: yes
    schedule:
        - '0 * * * *'
```

A *job block* is defined with the following parameters:

- **uri**: the identifier associated with this job's data.

Its like a path in user's data. Every user in Komlog can organize her time series in a structure like a filesystem. To identify each element in the structure we use what we call the **uri**.

We can nest our information in different levels using the dot character (.) For example, if we have uris *system.info.disk*, *system.info.disk.sda1* and *system.info.disk.sda2*, Komlog will nest them this way:

```
system
- info
   - disk
      - sda1
      - sda2
```

Important: An uri can be formed **only** with this characters:

- Capital or lowercase letters [A-Za-z]: ASCII characters from *A* to *z*.
- Numbers [0-9].
- Special characters:
  - Hyphen (-), underscore (_)

  - Dot (.)

An uri **cannot** start with dot (.).

- **command**: The command to execute.

Can be an operating system command or any script. komlogd will send the command/script standard output to Komlog.

Important: The command parameter cannot contain special command line characters like **pipes (|)** or **redirections (<,>)**. If you need them, create a script and include the command there.

- **enabled**: To enable or disable the job. Can take values *yes* or *no*.
- **schedule**: Sets the job execution schedule. Uses the classical UNIX cron format:

```
------------- minutes (0 - 59)
| ------------- hours (0 - 23)
| | ------------- day of month (1 - 31)
| | | ------------- month (1 - 12)
| | | | ------------- day of week (0 - 6) (sunday - saturday)
| | | | |
| | | | |
| | | | |
* * * * *
```

It accepts these special characters:

- Asterisk (*) to set every possible value of a group.
- Comma (,) to enumerate different values in a group.
- Slash (/) to set values of a division with zero remainder. So, for example, insted of setting minutes to *0,10,20,30,40,50* you can set **/10*.

The schedule parameter accepts as many elements as you need.

Every *job block* creates an independent process to manage the job execution, so they don't block each other. However, for security reasons, **komlogd will not execute more than one instance of each job in parallel**, so if you have a job that takes 10 minutes to complete and it is scheduled to execute every 5 minutes, the schedule will not be fulfilled.

2.2.1 Loading jobs from external files

You can tell komlogd to load jobs configuration from an external file instead of adding them directly to *komlogd.yaml*.

To achieve this:

1. Enable the external load option in *komlogd.yaml*:

```
- allow_external_jobs: yes
```

2. For each file, add an entry in *komlogd.yaml* like this one:

```
- external_job_file: <path_to_file>
```

Replacing *<path_to_file>* with the file's path. You can add as many *external_job_file* statements as you need.

2.3 Impulse methods configuration

Komlog allows the user to subscribe to any of his *uris* and receive notifications when new data is received on them.

An *impulse method* is the function that is executed when komlogd receives notifications about subscribed uris.

With *impulse methods* you can automate tasks, generate alarms, communicate with external services, analyze data in real time, and basically any task associated to events.

On chapter *Impulse methods* we explain how to create this type of functions.

To add a file with impulse methods to komlogd configuration we use the **impulses** block:

```
- impulses:
    enabled: yes
    filename: <path_to_file>
```

The *impulses block* parameters are:

- **enabled**: To enable or disable the block. Can take values *yes* or *no*.
- **filename**: Path to the impulses methods file. Path can be absolute or relative to the komlogd configuration directory.

You can add as many *impulses blocks* as you need.

2.4 Log configuration

komlogd logs configuration is stablished with the *logging* block in the configuration file. It has these default values:

```
- logging:
    level: INFO
    rotation: yes
    max_bytes: 10000000
    backup_count: 3
    dirname: log
    filename: komlogd.log
```

logging block parameters are:

- **level**: Sets the log level. Posible values are *CRITICAL, ERROR, WARNING, INFO, DEBUG, NOTSET*.
- **rotation**: Indicates if log file will be rotated when its size reaches *max_bytes* bytes. It accepts values *yes* or *no*.
- **max_bytes**: If log rotation is enabled, log file will be rotated when it reaches the size in bytes indicated by this parameter.
- **backup_count**: Number of log rotated files to keep on disk.
- **dirname**: Log file directory. Path can be absolute or relative to komlogd configuration directory.
- **filename**: log file name.

Once komlogd is configured, we can start it by executing the following:

```
komlogd &
```

CHAPTER 3

API

komlogd can be used as a library to add Komlog functionalities into your applications.

Note: komlogd uses asyncio internally, so your application must run an asyncio loop to be able to use it.

3.1 Session initialization

To stablish a Komlog session we need:

- Komlog user.
- RSA key.

Note: komlogd creates a RSA key during its *First execution*.

To open a session with Komlog, create a KomlogSession object with your username and private key and call the *login* method:

```
import asyncio
from komlogd.api import session, crypto

loop = asyncio.get_event_loop()

async def example():
    username='username'
    privkey=crypto.load_private_key('/path/to/key.priv')
    komlog_session=session.KomlogSession(username=username, privkey=privkey)
    await komlog_session.login()
    # At this point session is stablished.
    # If you need this task to wait in the loop while session is open:
    # await komlog_session.t_loop

    # To close the session:
    await komlog_session.close()

try:
    loop.run_until_complete(example())
except:
```

```
    loop.stop()
loop.close()
```

komlogd stablishes a web sockets connection with Komlog and internally manages reconnections.

Important: You need to authorize your agent previously on Komlog web, as explained in *komlogd authorization*.

3.2 Sending data to Komlog

Every metric you upload to Komlog is identified with its *uri*. Based on this uri, Komlog organize your metrics in a tree like structure. We call this structure *the user's data model*. Every metric in your data model can take different values at different timestamps. We call each of these values a *sample*.

You can create two types of metrics in Komlog:

- **Datasource**
- **Datapoint**

A metric is created automatically the first time you upload data to it and its type is stablished at this moment too. You cannot modify the metric type. If you need to modify it, you have to delete the metric and create it again with a different type.

Datasource metrics

A datasource is used to store texts. You can upload any text whose length is less or equal 130KB.

In this example you can see how to send a sample of a datasource metric:

```
import asyncio
import pandas as pd
from komlogd.api import session, crypto
from komlogd.api.model.orm import Datasource, Sample

loop = asyncio.get_event_loop()

async def send_datasource_sample():
    # connect to Komlog
    username='username'
    privkey=crypto.load_private_key('/path/to/key.priv')
    komlog_session=session.KomlogSession(username=username, privkey=privkey)
    await komlog_session.login()

    # prepare datasource sample
    uri='my_datasource'
    data='Datasource content'
    ts = pd.Timestamp('now',tz='Europe/Madrid')
    metric = Datasource(uri=uri)
    sample = Sample(metric=metric, ts=ts, data=data)

    # send sample and exit
    komlog_session.send_samples(samples=[sample])
    await komlog_session.close()

try:
    loop.run_until_complete(send_datasource_sample())
except:
```

```
    loop.stop()
finally:
    loop.close()
```

Datapoint metrics

A datapoint metric is used to store numerical values. You can use any type of numerical variables, like *int*, *float* or Decimal (we don't support values without numerical representation like *infinity*, *NaN*, etc).

In this fragment you can see how to send two samples associated to a pair of datapoint metrics:

```
import asyncio
import pandas as pd
from komlogd.api import session, crypto
from komlogd.api.model.orm import Datapoint, Sample

loop = asyncio.get_event_loop()

async def send_datapoint_samples():
    # connect to Komlog
    username='username'
    privkey=crypto.load_private_key('/path/to/key.priv')
    komlog_session=session.KomlogSession(username=username, privkey=privkey)
    await komlog_session.login()

    # prepare datapoint samples
    samples = []
    ts = pd.Timestamp('now',tz='Europe/Berlin')
    metric1 = Datapoint(uri='cpu.system')
    metric2 = Datapoint(uri='cpu.user')
    samples.append(Sample(metric=metric1, ts=ts, data=14.63))
    samples.append(Sample(metric=metric2, ts=ts, data=28.5))

    # send samples and exit
    komlog_session.send_samples(samples=samples)
    await komlog_session.close()

try:
    loop.run_until_complete(send_datapoint_samples())
except:
    loop.stop()
finally:
    loop.close()
```

Every metric in your *data model* is considered an independent time serie, so every sample you upload must be associated with a timestamp. **The timestamp is set by the user, so you can upload samples associated with a timestamp in the past or in the future.**

The timestamp can be any of these types:

- A *pandas.Timestamp* object.
- A *datetime.datetime* object.
- A ISO8601 formatted string.

It is mandatory to include the time zone and maximum precision is milliseconds.

3.3 Impulse methods

You can execute a function every time a metric is updated. We call this type of functions *impulse methods*.

To create impulse methods, simply add the *@impulsemethod* decorator to the function. You can pass the following parameters to the decorator:

- **uris**: list with metrics uris we want to subscribe our method to.
- **data_reqs**: it needs a DataRequirements object. With this parameter you set the method data requirements by uri.
- **min_exec_delta**: min time between execs. By default, komlogd will run the method every time a metric is updated. With this parameter you can tell komlogd to run it at most once in *min_exec_delta* interval. The parameter needs a pandas.Timedelta object.

Here you can see how to create an impulse method:

```
from komlogd.api.impulse import impulsemethod

@impulsemethod(uris=['cpu.system','cpu.user'])
async def example():
    print('hello komlog.')
```

In this example, every time metrics *cpu.system* and *cpu.user* are updated, komlogd will run the function *example*. **You can apply the @impulsemethod decorator to normal functions or coroutines**.

An impulse method can fire an update to any metric in our data model. For this, the method must return a dictionary with the samples to send to Komlog:

```
from komlogd.api.impulse import impulsemethod
from komlogd.api.model.orm import DataRequirements, Datasource, Sample

@impulsemethod(uris=['cpu.system','cpu.user'], data_reqs=DataRequirements(past_delta=pd.Timedelta('1h
def summary(ts, updated, data, others):
    result={'samples':[]}
    for metric in updated:
        int_data=data[metric][ts-pd.Timedelta('60 min'):ts]
        info=str(int_data.describe())
        stats = Datasource(uri='.'.join((metric.uri,'last_60min_stats')))
        sample = Sample(metric=stats, data=info, ts=ts)
        result['samples'].append(sample)
    return result
```

In this example we subscribe our summary function to metrics *cpu.system* and *cpu.user* and make some operations over their last hour data. Finally, we fire updates to metrics *cpu.system.last_60min_stats* and *cpu.user.last_60min_stats* with the data obtained from that operations. This method will run every time *cpu.system* or *cpu.user* are updated.

Here, we explain the last example in detail:

- **First, we apply to our *summary* function the decorator *@impulsemethod* with these parameters:**
  - **uris=['cpu.system','cpu.user']**. We set the metrics the method is subscribed to.
  - **data_reqs=DataRequirements(past_delta=pd.Timedelta('1h'))**. Here we indicate the method needs a data interval of 1h for each metric.
- ***summary* function receives some parameters (These parameters are optional. We have to declare them only if we need ther**
  - **ts**: pd.Timestamp object. The timestamp of the sample that fired the impulse method execution.

  - **updated**: list of metrics updated.
  - **data**: A dictionary whose keys are the subscribed metrics. Each value of the data dictionary is a pandas.Series object with the values of the metric.
  - **others**: list with metrics subscribed but not updated in this execution. updated + others will be all subscribed metrics.

- In the first method line, we declare variable result. this variable is a dict with the key *samples*. In this key we will store the samples to fire after the method execution.
- **Next, for every updated metric we do:**
  - Get the data from the last hour.
  - Store the result of applying the *describe* function to the data.(This function is from pandas module. It calculates some statistical values from a pandas.Series object).
  - Create a datasource whose *uri* is metric uri + *.last_60min_stats*, so we are creating **cpu.system.last_60_min_stats** and **cpu.user.last_60_min_stats**
  - Create a datasource Sample and set data and timestamp.
  - Append the sample to the samples key in the result dictionary.
- Return the result dictionary with the samples to send to Komlog.

You can stack the *impulsemethod* decorator as many times as you need. An impulse method will be created each time the decorator is applied. So for the previous example, if we want to create impulse methods for two groups of metrics, we can decorate the function once per group instead of creating two functions for the same operation.

```
@impulsemethod(uris=['host1.cpu.system','host1.cpu.user'], data_reqs=DataRequirements(past_delta=pd.T
@impulsemethod(uris=['host2.cpu.system','host2.cpu.user'], data_reqs=DataRequirements(past_delta=pd.T
def summary(ts, updated, data, others):
    ...
```

3.3.1 Working with remote uris

Users can share metrics through Komlog in real time with other users.

Note: You can share metrics through your Komlog configuration page. Keep in mind that metrics **will always be shared read only and recursively**, this means that if you share metric *cpu.system* every nested metric in the data model tree will be shared too, no matter if them already existed or not when the root metric was shared.

Sharing metrics read only means an *impulse method* cannot modify any remote metric, so if they return samples to update remote metrics, they will be dropped. Users **can only modify their own data model.**

With this functionality you can create applications based on distributed data models. The way to tell komlogd you want to subscribe to a remote uri is prepending the username to the local uri name:

```
remote_uri = 'user:uri'
```

For example, if user *my_friend* were sharing metrics *host1.cpu*, we could subscribe an impulse method to *host1.cpu.system* and *host1.cpu.user* this way:

```
@impulsemethod(uris=['my_friend:host1.cpu.system','my_friend:host1.cpu.user'], data_reqs=DataRequirer
def summary(ts, updated, data, others):
    ...
```

An impulse method can subscribe to both local and remote metrics:

```
uris = [
    'my_friend:host1.cpu.system',
    'my_friend:host1.cpu.user',
    'host1.cpu.system',
    'host1.cpu.user'
]

@impulsemethod(uris=uris, data_reqs=DataRequirements(past_delta=pd.Timedelta('1h')))
def summary(ts, updated, data, others):
    ...
```