Distlog Documentation

Release 0.1.0

Leo Noordergraaf

May 13, 2021

Contents

A logging system with an attitude for distributed software systems.

CHAPTER 1

Features

1.1 Distlog (Python package)

- Fully compatible with the standard Python logging package.
- Structured logging: Next to the ubiquitous log message, Distlog allows you to incorporate entire data sets in your log message.
- Scoped logging messages: Log messages can be structured in a tree like fashion. A task consisting of multiple subtasks, all producing log messages can be displayed as a tree the branches (subtasks) of which containing the leaves (log messages) can be folded and unfolded to show relevant or hide irrelevant messages.
- Scoped logging also works over process boundaries. A software system using a services architecture can create a coherent view of its execution where a service, possible executing on another host, is displayed as a subtask in the log message tree for a task.
- Implements a handler that uses ØMQ to send and collect log messages from all components of the distributed application.

1.2 Distlogd (daemon)

- Collects all log messages sent by Distlog over ØMQ.
- Uses a plugin architecture where each plugin gets to process all incoming messages. Plugins can then decide if and how to process a message.
- Contains predefined plugins to store the messages in a file, a MongoDB or Redis database or to publish them using ØMQ's PUB/SUB mechanism.
- Distlogd and its plugins are configurable using a YAML configuration file.

CHAPTER 2

Contents

2.1 Introduction

Distlog grew out of the frustration of working with the log files of a software system consisting of multiple cooperating processes, so-called microservices.

In order to track a request from the moment a client issued it all through its resolution and response consisted of searching through about half a dozen log files in multiple hosts all handling multiple requests concurrently.

2.1.1 Earlier attempts

Searching the net quickly showed that I wasn't the only frustrated programmer. Among others I found structlog, Logbook and Eliot. From them I learned about structured logging.

To me the phrase 'structured logging' has a double meaning. One the one hand it means that log messages can be enhanced, or even replaced by a dataset. On the other that the log messages themselves are structured, nested if you will. I will refer to this structure as scoping.

Both ideas appealed hugely to me and they cooperate nicely as the data structures are used to record the parent/child relationschip between log messages.

2.1.2 Still not happy

Yet all the mentioned solutions where lacking. Either it replaced the standard Python logging module used by nearly all packages. That would limit the usefullness of the logging messages since all messages from external packages are excluded and must be processed and correlated by hand as before. Or the solution will not work nicely over processes.

All solutions suffered that they focussed mostly on *generating* log messages but most of the work is in *processing* the messages. You want to filter and drill down on messages because you want to look only at that part of the system where an error occurs. Or log messages are used for other purposes such as collecting usage and performance data and you want those to be kept for a longer period and displayed in some dashboard whereas regular logging messages can be discarded after a few days.

2.1.3 Requirements

Thus grew the idea that a logging system for a distributed application is in itself quite a system. It needs to:

1. Generate (standard Python) log messages.
2. Scope log messages.
3. Extend log messages with additional data.
4. Collect those log messages in a central location.
5. Filter, store and/or redistribute those log messages.
6. Display log messages in a GUI or on a console and allow selection of relevant messages.
7. It should be able to influence message generation dynamically. So some parts of the system may produce DEBUG messages while others are less verbose.

The Distlog package implements the first three items, Distlogd the next two. There is still some work to do.

2.2 Tutorial

This tutorial tries to give you a taste of what it will be like to use this system. At the same time it also allows to collect and refine my ideas. The contents of this section is far from stable.

Loading and initializing distlog:

```
1 import logging
2 import distlog
3
4 logger = logging.getLogger()
5 handler = distlog.ZmqHandler('tcp://localhost:5010')
6 handler.setFormatter(distlog.JSONFormatter())
7 logger.addHandler(handler)
8 logger.setLevel(logging.INFO)
```

On line 5 the ØMQ handler is created and intialized. It will bind to the Distlogd daemon listening on port 5010 on the local host.

Line 6 sets the formatter to the JSON encoder. You may also choose to use the faster pickle encoder.

Using distlog:

```
10 def main():
11     with distlog.task('toplevel', user='leo') as job:
12         print('into task')
13         logger.info('into task')
14         with distlog.to('subtask', arg=42) as job:
15             print('into subtask')
16             logger.info('into task')
17             job.success('subtask done')
18     print('all done')
19     logger.info('all done')
20
21 if __name__ == '__main__':
22     main()
```

Attitude

I said Distlog has something of an attitude. And here it is. It assumes that you structure your program by stating its purpose, its task, and then proceed by implementing this task using smaller subtasks:

```
with task('BEING IMPORTANT'):
    with to('appear important'):
        with to('dress up'):
            pass
```

In other words each subtask of your program is encapsulated in a Distlog context. –I use context and scope interchangeably to mean the same thing.

On line 11 the task is defined. This is the outermost scope of the logging tree. The string becomes part of the task initiation termination messages. Any positional arguments are assumed to be format parameters for the message just as with the regular Python logging system. But all keyword arguments are stashed away and added to all log messages that are generated in this scope. Here that is only happens on line 13.

On line 14 a new scope is created as a child of the toplevel scope of line 11. The set of keyword arguments replaces those of the outer scope. You can always find them through the encompassing scope.

On line 17 the inner scope is given a new logging message to use if the subtask completes without an exception.

Finally on line 19 a log message is produced which is outside the toplevel scope.

When this program is run the console will display:

```
into task
into subtask
all done
```

Over the ØMQ socket the following messages are sent (pretty printed):

```
{
    "context": {
        "key": "0@e252a11f-d33d-483d-ba08-bc8f642b2f10",
        "user": "leo"
    },
    "filename": "example.py",
    "funcName": "main",
    "stack_info": null,
    "args": null,
    "process": 6156,
    "hostname": "obelix",
    "msecs": 148.06699752807617,
    "message": "toplevel",
    "name": "root",
    "module": "example",
    "thread": 139753641113344,
    "msg": "toplevel",
    "lineno": 13,
    "threadName": "MainThread",
    "exc_text": null,
    "exc_info": null,
    "levelno": 20,
    "asctime": "2018-04-18 23:03:19,148",
    "relativeCreated": 379328.7272453308,
    "levelname": "INFO",
```

(continues on next page)

(continued from previous page)

```
    "processName": "MainProcess",
    "created": 1524085399.148067,
    "pathname": "/home/leo/src/distlog/example.py"
}
```

This is basically Python's LogRecord structure. It has an extra field *context* containing the additional keyword argument and a *key* field which is used to correlate the messages.

The *key* field consists of three parts:

- message sequence number
- unique toplevel scope identification
- optional subscope sequence number

The other JSON messages are:

```
{
    "context": {
        "key": "1@e252a11f-d33d-483d-ba08-bc8f642b2f10",
        "user": "leo"
    },
    "filename": "example.py",
    "funcName": "main",
    "stack_info": null,
    "args": null,
    "process": 6156,
    "hostname": "obelix",
    "msecs": 824.9077796936035,
    "message": "into task",
    "name": "root",
    "module": "example",
    "thread": 139753641113344,
    "msg": "into task",
    "lineno": 15,
    "threadName": "MainThread",
    "exc_text": null,
    "exc_info": null,
    "levelno": 20,
    "asctime": "2018-04-18 23:06:30,824",
    "relativeCreated": 571005.5680274963,
    "levelname": "INFO",
    "processName": "MainProcess",
    "created": 1524085590.8249078,
    "pathname": "/home/leo/src/distlog/example.py"
}
```

```
{
    "context": {
        "key": "0@e252a11f-d33d-483d-ba08-bc8f642b2f10/1",
        "arg": 42
    },
    "filename": "example.py",
    "funcName": "main",
    "stack_info": null,
    "args": null,
    "process": 6156,
```

(continues on next page)

(continued from previous page)


```
    "hostname": "obelix",
    "msecs": 113.48962783813477,
    "message": "subtask",
    "name": "root",
    "module": "example",
    "thread": 139753641113344,
    "msg": "subtask",
    "lineno": 16,
    "threadName": "MainThread",
    "exc_text": null,
    "exc_info": null,
    "levelno": 20,
    "asctime": "2018-04-18 23:07:18,113",
    "relativeCreated": 618294.1498756409,
    "levelname": "INFO",
    "processName": "MainProcess",
    "created": 1524085638.1134896,
    "pathname": "/home/leo/src/distlog/example.py"
}
```

```
{
    "context": {
        "key": "1@e252a11f-d33d-483d-ba08-bc8f642b2f10/1",
        "arg": 42
    },
    "filename": "example.py",
    "funcName": "main",
    "stack_info": null,
    "args": null,
    "process": 6156,
    "hostname": "obelix",
    "msecs": 585.9096050262451,
    "message": "into task",
    "name": "root",
    "module": "example",
    "thread": 139753641113344,
    "msg": "into task",
    "lineno": 18,
    "threadName": "MainThread",
    "exc_text": null,
    "exc_info": null,
    "levelno": 20,
    "asctime": "2018-04-18 23:07:35,585",
    "relativeCreated": 635766.569852829,
    "levelname": "INFO",
    "processName": "MainProcess",
    "created": 1524085655.5859096,
    "pathname": "/home/leo/src/distlog/example.py"
}
```

```
{
    "context": {
        "key": "2@e252a11f-d33d-483d-ba08-bc8f642b2f10/1",
        "arg": 42
    },
    "filename": "example.py",
```


(continues on next page)

(continued from previous page)


```
    "funcName": "main",
    "stack_info": null,
    "args": null,
    "process": 6156,
    "hostname": "obelix",
    "msecs": 411.38386726379395,
    "message": "subtask done",
    "name": "root",
    "module": "example",
    "thread": 139753641113344,
    "msg": "subtask done",
    "lineno": 19,
    "threadName": "MainThread",
    "exc_text": null,
    "exc_info": null,
    "levelno": 20,
    "asctime": "2018-04-18 23:08:13,411",
    "relativeCreated": 673592.0441150665,
    "levelname": "INFO",
    "processName": "MainProcess",
    "created": 1524085693.4113839,
    "pathname": "/home/leo/src/distlog/example.py"
}
```


Note that the message shows the contents of the success() parameters.


```
{
    "context": null,
    "filename": "example.py",
    "funcName": "main",
    "stack_info": null,
    "args": null,
    "process": 6156,
    "hostname": "obelix",
    "msecs": 740.2544021606445,
    "message": "all done",
    "name": "root",
    "module": "example",
    "thread": 139753641113344,
    "msg": "all done",
    "lineno": 21,
    "threadName": "MainThread",
    "exc_text": null,
    "exc_info": null,
    "levelno": 20,
    "asctime": "2018-04-18 23:08:52,740",
    "relativeCreated": 712920.9146499634,
    "levelname": "INFO",
    "processName": "MainProcess",
    "created": 1524085732.7402544,
    "pathname": "/home/leo/src/distlog/example.py"
}
```


Outside of any context so the *context* field is null/None.

2.3 Organization

2.4 Distlog

The distlog package is the library you add to your code. It is used in combination with the regular Python `logging` module.

`distlog.`**`import_task`**(*_id*, *msg*, **args*, ***kwargs*)

Link task to external parent.

Let this task be a subtask of a task in a different process. Allows you to see the sequence of calls and log messages over process boundaries.

_id is obtained in the calling process through the *get_foreign_task()*. How the value is transferred from the caller to the callee is application defined. That depends on the application and communication protocols.

```
from distlog import import_task
id = get_foreign_task_id()
myhost = 'sample.example.com'

with import_task(id, 'continued processing', host=myhost):
    # do something interesting
```

Parameters

- **`_id`** (*string*) – task id calculated by the foreign parent task
- **`msg`** (*string*) – log message used with entering (and optionally when leaving the task
- **`args`** (*list*) – parameters for the log message
- **`kwargs`** (*dict*) – key/value pairs forming the log message context.

Return type *Task*

`distlog.`**`task`**(*msg*, **args*, ***kwargs*)

Create a toplevel task.

Creates a new toplevel context. You would use this when a new activity is started. A task is generally composed from smaller subtasks.

```
from distlog import task
with task('counting up to %d', 10, job='count up', until=10):
    for i in range(10):
        print(i)
```

Parameters

- **`msg`** (*string*) – state the goal of this program
- **`args`** (*list*) – parameters for the goal
- **`kwargs`** (*dict*) – key/value context for the log messages

Return type *Task*

`distlog.`**`to`**(*msg*, **args*, ***kwargs*)

Create a subtask.

Encapsulates a part of a larger program. You typically use this to delineate a program section that performs a specific job. Usually this means that it encapsulates a function.

Todo Create a decorator to easily encapsulate a function.

```
from distlog import to
val = 10

with to('print int', arg=val):
    print(val)
```

Parameters

- **msg** (*string*) – defines the goal of this subtask
- **args** (*list*) – parameters for the goal
- **kwargs** (*dict*) – context for log messages of this task

Return type *Task*

class distlog.**JSONFormatter**(*fmt=None*, *datefmt=None*)
Formatter to convert to JSON format.

encoding
Describe the encoding used.

Return string encoding indicator

format(*record*)
Format to JSON.

Use the logging Formatter to convert the LogRecord data to JSON for network transport.

Parameters **record** – LogRecord instance

Returns JSON encoded record contents.

class distlog.**PickleFormatter**(*fmt=None*, *datefmt=None*)
Formatter to convert to pickle format.

encoding
Describe the encoding used.

Return string encoding indicator

format(*record*)
Format to pickle.

Use the logging Formatter to convert the LogRecord data to pickle format or network transport.

Parameters **record** – LogRecord instance

Returns pickled record contents.

class distlog.**ZmqHandler**(*endpoint*, *context=None*, *system='P'*)
0MQ transport implementation.

emit(*record*)
Do whatever it takes to actually log the specified logging record.

setFormatter(*fmt*)
Set the formatter for this handler.

set_topic(*encoding*)
Set message topic elements.

Creates the topic strings for log and performance messages. :param system: the system topic :param encoding: the encoding topic

class distlog.**Task**(*_id*, *msg*, **args*, ***kwargs*)
Define the scope and context for other tasks and log messages.

A *Task* defines the context for other tasks and log messages. It is a context manager that is ususally created by calling one of the functions *task()* or *to()* where *task()* creates a new top-level context and *to()* creates a subtask.

The only real difference between the two is the way the id parameter is defined. A *task()* creates a UUID as id and *to()* creates a sequence number as id. *import_task()* is an alternative for *to()*, to be used when the subtask resides in another process.

When the __enter__() method of the *Task* is called it will emit a log message at INFO level signalling the begin of the scope. Thereafter all log messages will belong to the same scope until a subtask is created with the *to()* function or until the *with* section terminates.

When the __exit__() method of the *Task* is called another log message at the INFO level signals the completion of the scope. The cause of leaving the *with* section, with an exception or normally, controls which log message is produced.

Parameters

- **_id** (*string*) – Either absolute (task) or relative (to) task id.
- **msg** (*string*) – Task description, may contain % formatting.
- **args** (*list*) – Parameters for the msg parameter.
- **kwargs** (*dict*) – Context definition. You may provide as many named parameters as needed. They are stored as key/value pairs in the context and are added to the *LogRecord* when it is created.

bind(***kwargs*)
Add key/value pairs to the context.

Add the key/value pairs in kwargs to the dataset that is eventually returned by the context property.

Parameters **kwargs** (*dict*) – dict with key/value pairs

context
Assemble the context.

The assembled context is added to the *LogRecord* instance as the *context* attribute. The assembly consists of the key identifying this log entry with the data provided when creating the context and any data added to it by the *bind()* function.

Vartype dict containing the context data.

get_foreign_task()
Determine task id for a foreign task.

The returned ID can be passed over the network in an undefined manner and used on the other side as the first parameter for the function *import_task()*. Doing so will link both tasks (although in separate processes) toghether in one related set of log messages.

Return string child task identification string.

get_next_task()
Identify the new subtask.

Every subtask spawned from this scope is identified by the current scopes identifier suffixed by the subtask sequence number. This method increments that sequence number to create a new scope.

Return int subtask sequence number.

id
Property produces the context's unique id.

parent
Parent id if available.

Provides the value of the parents *id* property or *None* if there is no parent.

Vartype string Id of the parent component.

success(*msg*, **args*)
Set success report message.

When the context manager is terminated successfully and *success()* has been called on the context then the *msg* is used to report the status in a log message.

Parameters

- **msg** (*string*) – The log message to display
- **args** (*list*) – optional format parameters for the msg

class distlog.**LogContext**
Log message context.

The log message context is a stack of *Task* entries. Tasks form the context for individual log messages. Tasks can be constructed from subtasks which is why a stack structure is required.

The *LogContext* itself is a class with only a single instance which acts as a singleton and is defined as a module global in this file.

pop()
Remove top element of the stack.

Pop and return the topmost element of the stack.

Return type *Task* the removed element.

push(*action*)
Push element onto the stack.

Parameters **action** (*Task*) – item to add to the stack.

top
Produce element on top of stack.

Vartype *Task* the element on top of the stack.

class distlog.**LogRecord**(*name*, *level*, *pathname*, *lineno*, *msg*, *args*, *exc_info*, *func=None*)
LogRecord replacement.

This class replaces the standard LogRecord class. It provides a single change: when an instance is created the instance is extended with a context attribute containing the current scope's context.

2.5 Distlogd

2.6 Plugins

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index