UDPLog Documentation

Release 0.1.1

Ralph Meijer

February 15, 2014

UDPLog is a system for emitting application log events via UDP and shipping them via RabbitMQ or Scribe for further processing. The idea is that the applications sends its structured log events to a dedicated shipping daemon on the same machine, which in turn passes it on to one or more remote services. As this uses UDP, emitting events is non-blocking fire-and-forget. A system like Logstash can then be used to process and store log events.

Warning: The U in UDPLog stands for **unreliable**, and that's just what it is. Log messages are delivered on a best-effort basis and are not guaranteed in any way, even though the default mode is to send logs to a daemon running on the same machine.
Do **not** use UDPLog for any data that must be reliable, such as any information used for billing of any sort!

Contents:

CHAPTER 1

Python Client support

1.1 Using the UDPLog client directly

The Python UDPLog client is in `udplog.udplog`. Use like so:

```
from udplog import udplog
logger = udplog.UDPLogger()
logger.log('some_category', {'a_key': 'a_value', 'another_key': 17})
```

The dictionary passed as the second arg must be safe for JSON encoding.

1.2 Using the Python logging facility

This is the preferred method when the log events should also appear in local log files. Every Python source file will add this at the top:

```
import logging

logger = logging.getLogger(__name__)
```

This will set up a local logger that carries the module path as the logger name.

Then, `logger` is a `logging.Logger` on which you can call methods like `debug()`, `info()` and `error()`:

```
logger.info("Function %(func)s took %(time_elapsed)s",
            {'func': func,
             'time_elapsed': time.time() - start_time})
```

The logged message will be passed through the Python logging facility, to be delivered to log handlers. Using a format string and a dictionary with values, instead of a pre-formatted log message, makes those values individually available to the log handlers. For `UDPLogHandler`, explained below, those values become available as keys in the JSON sent out. Then, when the log events are sent on to Logstash, they appear as fields that can be used for filtering and queries.

When an exception occurs that needs to be logged, you can use `exception()` from an exception handler. This will allow log handlers to add tracebacks to the logged entries. In this case, `UDPLogHandler` will add three fields to the log event: `excText`, `excType` and `excValue`. Respectively, they hold the rendered traceback, the exception type and the exception value (usually the arguments passed to the exception when raised):

```
a = {}
try:
    b = a['foo']
```

```
except:
    log.exception("Oops, no %(bar)s", {'bar': 'foo'})
```

`UDPLogHandler` further adds fields for the filename and the line number where the log event was created (`filename` and `lineno`), as well as the function name (`funcName`) and the log level and logger name (`logLevel` and `logName`).

The log handler usually only has to be setup once per application, for example in the application's `main` function:

```
logging.basicConfig()
root = logging.getLogger()
root.setLevel(logger.INFO)
root.addHandler(udplog.UDPLogHandler(UDPLogger(), 'python_logging'))
```

This will set up default logging to `stderr`, set the minimum log level to `INFO` and then add a handler for logging to UDPLog. The second argument passed to the handler is the UDPLog category. You can override this on individual log events by adding a `category` field in the dictionary passed as the second argument to the log methods.

The handler also supports the `extra` keyword argument to the logger methods, adding the values to the emitted dictionary. The logging module has the very useful `LoggerAdapter` to wrap a regular logger to add extra data to every log event.

A complete Python logging example:

```
"""
Example of using the standard logging facility to send events to udplog.
"""

import logging
import socket
import warnings

from udplog.udplog import UDPLogger, UDPLogHandler

# Get a logger in the idiomatic way.
logger = logging.getLogger(__name__)

# Set up logging to stdout
logging.basicConfig(level=logging.DEBUG)

# Capture warnings, too.
logging.captureWarnings(True)

# Add the UDPLog handler to the root logger.
udplogger = UDPLogger(defaultFields={
                          'appname': 'example',
                          'hostname': socket.gethostname(),
                          })
root = logging.getLogger()
root.setLevel(logging.DEBUG)
root.addHandler(UDPLogHandler(udplogger, category="python_logging"))

def main():
    logger.debug("Starting!")
    logger.info("This is a simple message")
    logger.info("This is a message with %(what)s", {'what': 'variables'})

    extra_logger = logging.LoggerAdapter(logger, {'bonus': 'extra data'})
    extra_logger.info("Bonus ahead!")
```

```
36
37      a = {}
38      try:
39          print a['something']
40      except:
41          logger.exception("Oops!")
42
43      warnings.warn("Don't do foo, do bar instead!", stacklevel=2)
44
45  main()
```

The call to `exception()` results in this event dictionary:

```
{
  "appname": "example",
  "category": "python_logging",
  "excText": "Traceback (most recent call last):\n
                File \"doc/examples/python_logging.py\", line 39, in main\n
                  print a['something']\n
              KeyError: 'something'",
  "excType": "exceptions.KeyError",
  "excValue": "'something'",
  "filename": "doc/examples/python_logging.py",
  "funcName": "main",
  "hostname": "localhost",
  "lineno": 41,
  "logLevel": "ERROR",
  "logName": "__main__",
  "message": "Oops!",
  "timestamp": 1379508311.437895
}
```

1.3 Using the Twisted logging system

Twisted has its own logging system in `twisted.python.log` and `udplog.twisted.UDPLogObserver` can be set up to send all logged events onto a UDPLog server. It has special support for rendering exceptions and warnings. See `UDPLogObserver.emit` for details.

The following Twisted logging example sets up the log observer and uses the Twisted logging system to emit a few log events:

```
1   """
2   Example of using the Twisted logging facility to send events to udplog.
3   """
4
5   import sys
6   import warnings
7
8   from twisted.python import log
9
10  from udplog.udplog import UDPLogger
11  from udplog.twisted import UDPLogObserver
12
13  # Set up logging to stdout
14  log.startLogging(sys.stdout)
15
16  # Set up the udplog observer
```

```
17 udplogger = UDPLogger()
18 observer = UDPLogObserver(udplogger, defaultCategory='twisted_logging')
19 log.addObserver(observer.emit)
20
21 def main():
22     log.msg("Starting!")
23     log.msg("This is a simple message")
24     log.msg(format="This is a message with %(what)s", what='variables')
25
26     a = {}
27     try:
28         print a['something']
29     except:
30         log.err(None, "Oops!")
31
32     warnings.warn("Don't do foo, do bar instead!", stacklevel=2)
33
34 main()
```

The call to `log.err` has `None` as the first argument, so that the most recent exception is retrieved from the execution context. Alternatively, you can pass an exception instance or `failure.Failure`. The second argument is the *why* of the log event, and ends up in the `message` field of the event dictionary:


```
{
  "category": "twisted_logging",
  "excText": "Traceback (most recent call last):\n
                File \"doc/examples/twisted_logging.py\", line 34, in <module>\n
                  main()\n
              --- <exception caught here> ---\n
                File \"doc/examples/twisted_logging.py\", line 28, in main\n
                print a['something']\nexceptions.KeyError: 'something'\n",
  "excType": "exceptions.KeyError",
  "excValue": "'something'",
  "isError": true,
  "logLevel": "ERROR",
  "message": "Oops!",
  "system": "-",
  "timestamp": 1379507871.564469
}
```


The warning is rendered as follows:


```
{
  "category": "twisted_logging",
  "filename": "doc/examples/twisted_logging.py",
  "isError": false,
  "lineno": 34,
  "logLevel": "WARNING",
  "message": "Don't do foo, do bar instead!",
  "system": "-",
  "timestamp": 1379507871.564662,
  "warningCategory": "exceptions.UserWarning"
}
```

CHAPTER 2

Code Examples

- python_logging.py — An example of using UDPLog with the Python logging facility.
- python_twisted.py — An example of using UDPLog with the Twisted logging system.

CHAPTER 3

Protocol

A log event a combination of a category identifier (ASCII, matching the regular expression `^[0-9A-Za-z_]+$`) and a set of name/value pairs. The UDP wire protocol represents an event as a single datagram composed of the the category, a colon character, an optional whitespace character and the name/value pairs rendered as a JSON object:

```
some_category: {"a_key": "a_value", timestamp: "1379002018.000"}
```

CHAPTER 4

What to log and what to call it

Log everything. It's better to over-log than under-log.

In general, it is better to have to fewer distinct categories, and to have multiple types of entries in the same category if they have common fields. Using ElasticSearch and Logstash, you can then define a mapping for each category. To distinguish the events in the same category, you can add another field like `event`.

In general, an application shouldn't have more than a few categories, and categories can span multiple applications.

The UDPLog libraries already adds a timestamp to each event. You do not need to add a timestamp to your logs, unless you want to record the exact time your event happened as opposed to the time the log was created. In that case, set the `timestamp` field.

All times should be expressed in seconds, not micro or milliseconds. Generally, you'll want a floating point number of seconds. Timestamps are expressed as floating point seconds since the UNIX epoch.

CHAPTER 5

Emitting log events

There are a several ways to emit log events from applications:

- With direct calls to the logger.
- Via the Python logging facility.
- Via the Twisted logging system.

See *client* for details.

CHAPTER 6

Receiving log events

To ship log events to Scribe and/or RabbitMQ, there is Twisted-based support for receiving UDPLog events and passing them to those systems. This is exposed via a `twistd` plugin called `udplog`.

Pass all events to a local Scribe service:

```
twistd udplog --scribe-host=localhost
```

Pass all events to a RabbitMQ server, using the exchange named *logs*:

```
twistd udplog --rabbitmq-host=10.0.0.2 --rabbitmq-exchange=logs
```

For a full list of command line options, run:

```
twistd udplog --help
```

CHAPTER 7

Testing

To capture logs during application development, run a UDPLog daemon. This will print to the console all messages it receives:

```
python -mudplog.udplog
```

Or, using the `twistd` plugin:

```
twistd -n udplog --verbose
```

CHAPTER 8

Indices and tables

- *genindex*
- *search*