

Monitoring Python Applications With Kieker

Reiner Jung¹, Sven Gundlach¹, Serafim Simmonov¹ and Wilhelm Hasselbring¹

¹Kiel University, Christian-Albrechts-Platz 4, 24103 Kiel, Germany

Keywords

Application Level Monitoring, Kieker, Instrumentation, Python

Python is a widely used programming for applications, web services, and, especially, in scientific computing and data science. In the context of our project OceanDSL, which aims to provide DSLs for ocean system models, we need to comprehend existing scientific software based on static and dynamic code analysis. Thus, we decided to provide monitoring support for Python utilizing the existing Kieker analysis toolchain. As the code base is rather extensive, manually injecting probes is not a viable solution. Thus, our implementation relies on code weaving approaches.

The Kieker Language Pack for Python follows, in principle, the Kieker architecture for monitoring with a reduced feature set [1]. It comprises (a) event types, (b) code to control probes and data storage, (c) probes to instrument the code, and (d) a techniques to introduce probes into a program without modifying the code manually.

Event types for Python can be generated utilizing the Kieker instrumentation record language (IRL) which we extended to support Python [2]. Currently, the event types consist, like their Java counterparts, of a set of constants implementing default values, attributes, a constructor and a serialization method which utilizes a serialization helper to support multiple formats (cf. Listing 1). As the language pack is only used to monitor and log events in Python applications, it does not support features used to deserialize and manage events. However, this can be added later, if needed.

Listing 1: Simplified OperationExecutionRecord

```
class OperationExecutionRecord:
    __NO_OPERATION_SIGNATURE__ = "noOperation"

    def __init__(self, operation_signature, trace_id, tin, tout):
        self.operation_signature = (self.__NO_OPERATION_SIGNATURE__
                                     if operation_signature is None
                                     else operation_signature)
        self.trace_id = trace_id
        self.tin = tin
        self.tou = tout
```


SSP'21: Symposium on Software Performance, November 09–10, 2021, Leipzig, Germany

✉ reiner.jung@email.uni-kiel.de (R. Jung); sven.gundlach@email.uni-kiel.de (S. Gundlach);
stu126367@mail.uni-kiel.de (S. Simmonov); hasselbring@email.uni-kiel.de (W. Hasselbring)

🆔 0000-0002-5464-8561 (R. Jung); 0000-0003-4060-2754 (S. Gundlach); 0000-0001-6625-4335 (W. Hasselbring)



© 2021 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

```
def serialize(self, serializer):
    serializer.put(self.operation_signature)
    serializer.put(self.tin)
    serializer.put(self.tout)
```

Logging, time keeping, and probe control are handled by a monitoring controller utilizing a writer and a time controller following the Kieker architecture in a simplified adaptation. The controller provides all the necessary functionality for monitoring probes. The writer controller supports logging into files and transfer via TCP. The file logging serializes event types following the Kieker text file format. The TCP logging utilizes the Kieker binary logging format, which is also used by the Kieker Java implementation and Kieker Language Pack for C and Fortran. The latter logging feature allows to transfer data to a dedicated logging computer and either store the log data with the Kieker collector tool (the replacement for the Kieker data bridge) or feed the information directly into an analysis.

The probes follow the same general structure implementing advices which can be applied manually or with different methods automatically. We implemented a basic set of probes and tooling supplemented with documentation. To apply the probes, users can rely on our own weaving technique which utilizes Python's own weaving feature. This has the benefit that no additional libraries must be used. Alternatively, Python `aspectlib` can be used which provides convenient functions to weave probes into Python.¹

While the Kieker Language Pack for Python is in an early stage, we applied it successfully to the Kieker bookstore example. The tooling is available on the github page of Kieker.² Our next steps are to complete the implementation to support trace and data flow monitoring for Python. Furthermore, we aim to create an installation package for Python supporting pip to reduce hurdles for users including ourselves, and apply it to various tools used in data science including Jupyter notebooks.

Acknowledgments

Funded by the Deutsche Forschungsgemeinschaft (DFG – German Research Foundation), grant no. HA 2038/8-1 – 425916241.

References

- [1] W. Hasselbring, A. van Hoorn, Kieker: A monitoring framework for software engineering research, *Software Impacts* 5 (2020). doi:10.1016/j.simpa.2020.100019.
- [2] R. Jung, C. Wulf, Advanced typing for the Kieker instrumentation languages, in: *Symposium on Software Performance 2016*, 2016. URL: <http://oceanrep.geomar.de/34626/>.

¹Aspectlib <https://pypi.org/project/aspectlib/>

²Kieker Language Pack for Python <https://github.com/silvergl/kieker-lang-pack-python/>