Software Performance Anti-Patterns Observed and Resolved in Kieker

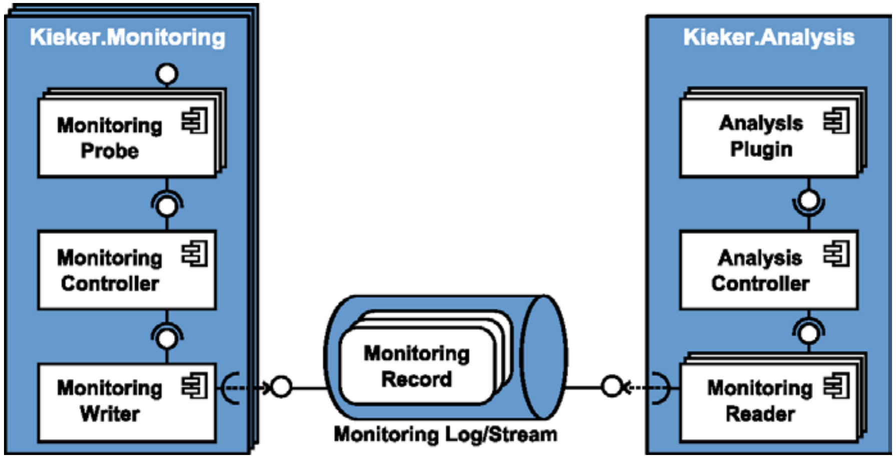
Symposium on Software Performance 2015

Christian Wulf and Wilhelm Hasselbring

06.11.2015

Software Engineering Group
Kiel University, Germany

Kieker‘s Software Architecture



- Low monitoring overhead

MooBench

- Fast Pipe-and-Filter-based analyses (migration currently in progress)

TeeTime

Software Performance Anti-Patterns

- Problem solutions which have a negative impact on the performance
- Pattern:
  - name
  - problem description of the solution
  - better solution

Excerpt of 14 anti-patterns (Smith et al. [3])

- “god” Class
- Unnecessary Processing
- Excessive Dynamic Allocation

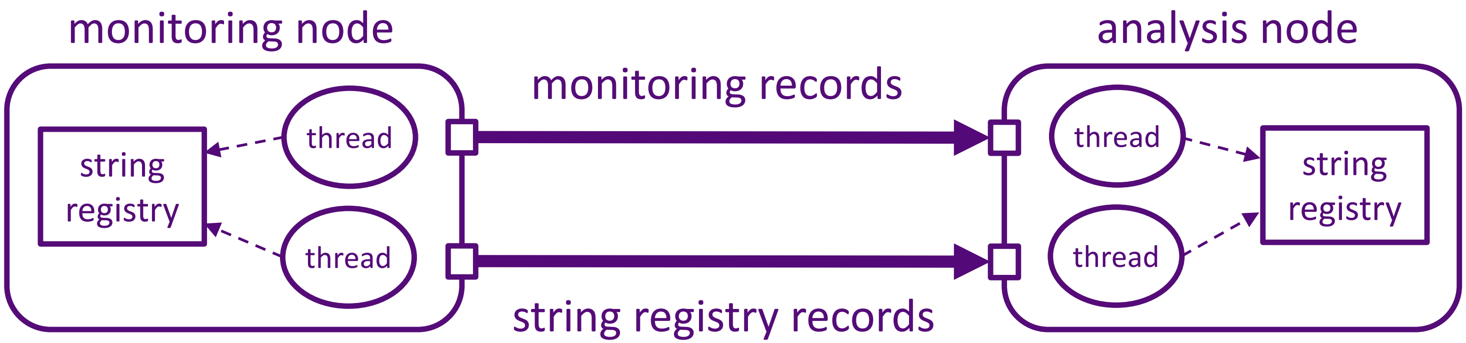
Agenda

- Introduction
- PAA #1: Parallelizing Sequential Dependencies
- PAA #2: Reflection-based Record Reconstruction
- PAA #3: Exception-based Buffer Underflow Detection
- Conclusion

PAA #1: Parallelizing Sequential Dependencies

Christian-Albrechts-Universität zu Kiel

Context (Kieker 1.12 and below):

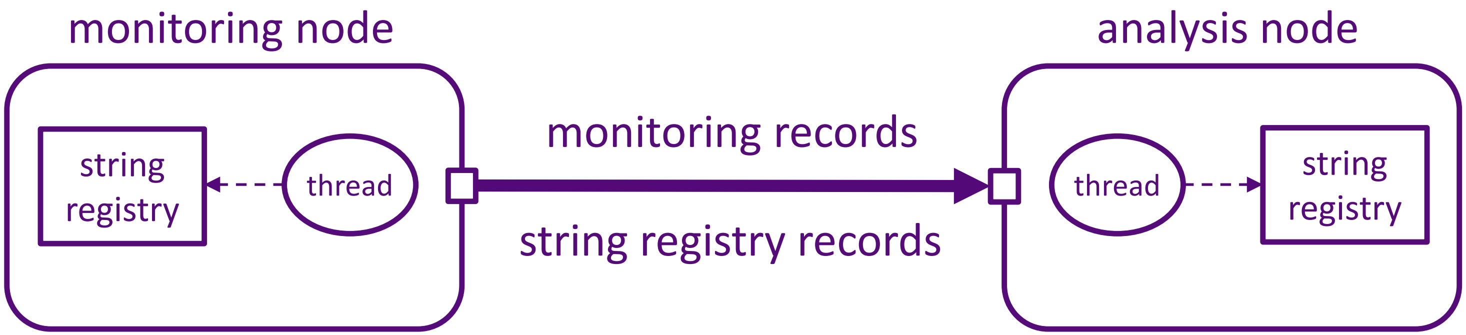


Issues:

- Two TCP connections
  => higher maintenance effort and higher security risk
- Thread synchronization (via string registry)
  => higher communication effort
- (Blocking) wait if a monitoring record arrives before its string registry records
  => reduced throughput

PAA #1: Parallelizing Sequential Dependencies

Our solution:



Approach:

- First, serializes all string registry records
- Then, serializes the record

Benefits:

- Only one TCP connection
- No thread synchronization required
  => Unsynchronized string registry is sufficient
- No waits required

PAA #2: Reflection-based Record Reconstruction

Context (Kieker 1.10 and below):

```
int classId = buffer.getInt();
recordClassName = stringRegistry.get(classId);
record = AbstractMonitoringRecord.createFromByteBuffer(
       recordClassName, buffer, stringRegistry);
```

Major issue:

- Reflective invocation of the record’s constructor
  => Slow, especially due to the frequent invocations[1]

[1] http://docs.oracle.com/javase/tutorial/reflect/index.html

PAA #2: Reflection-based Record Reconstruction

Our solution:

```
int classId = buffer.getInt();
recordClassName = stringRegistry.get(classId);
recordFactory = cachedRecordFactoryCatalog.get(recordClassName);
record = recordFactory.create(buffer, stringRegistry);
```

```
return new ConcreteRecord(..)
```

Approach:

- Introduction of a record factory per record type
- Reflective search only once for each record factory

  ⇒ Caches subsequent accesses in a map

Benefits:

- Direct invocation via Java’s keyword `new`

  => Fast record construction

PAA #3: Exception-based Buffer Underflow Detection

Context (Kieker 1.12 and below):

```
try {
       // save buffer's current position
       reconstruct(buffer);
} catch (BufferUnderflowException e) {
       // refill buffer
       // reset buffer's position
}
```

Issues:

- Creation of a new exception object
- Resolution of the current stacktrace

  => Slow and not used at all

PAA #3: Exception-based Buffer Underflow Detection

Our solution:

```
// save buffer's current position
boolean success = reconstruct(buffer);
if (!success) {
        // refill buffer
        // reset buffer's position
}
```

Approach:

- Check whether the buffer has enough bytes left for the next record
- Return a boolean value indicating a buffer refill

Benefits:

- No creation of an exception
- No stacktrace resolution

} Fast buffer underflow detection

Conclusion

- PAA #1: Parallelizing Sequential Dependencies
- PAA #2: Reflection-based Record Reconstruction
- PAA #3: Exception-based Buffer Underflow Detection

Kieker

http://kieker-monitoring.net

TeeTime

http://teetime.sourceforge.net

Future work:

- Avoid redundant information in before/after record
- Avoid frequent record construction/destruction scenarios (reduce GC time)

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-oriented Software. Prentice Hall, 1995.

[2] A. Koenig. Patterns and antipatterns. In The Patterns Handbooks. Cambridge University Press, 1998.

[3] C. U. Smith and L. G. Williams. More new software performance antipatterns: Even more ways to shoot yourself in the foot. In Proc. of the Int. CMG Conference, 2003.

[4] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In Proc. of the ICPE, 2012.

[5] J. Waller, F. Fittkau, and W. Hasselbring. Application performance monitoring: Trade-off between overhead reduction and maintainability. In Proc. of the Symposium on Software Performance, 2014.

[6] M. Wooldridge and N. R. Jennings. Pitfalls of Agent-oriented Development. In Proc. of the AGENTS, 1998.