Kieker’s Software Architecture

- Low monitoring overhead
- Fast Pipe-and-Filter-based analyses (migration currently in progress)
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
• Introduction

• PAA #1: Parallelizing Sequential Dependencies
• PAA #2: Reflection-based Record Reconstruction
• PAA #3: Exception-based Buffer Underflow Detection
• Conclusion
Context (Kieker 1.12 and below):

![Diagram of monitoring and analysis nodes with string registry records]

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
Context (Kieker 1.10 and below):

```java
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](http://docs.oracle.com/javase/tutorial/reflect/index.html)
Our solution:

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

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
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
Our solution:

```java
// 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

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


