

The Time Measurement Pattern in the JVM

David Georg Reichelt 
Lancaster University Leipzig /
URZ Leipzig

Shinhyung Yang 
Software Engineering
Kiel University

Wilhelm Hasselbring 
Software Engineering
Kiel University

Abstract

High performance applications-like distributed tracing-require collecting timing information for all observed events. To obtain usable timing information across distributed nodes, absolute timestamps are required. When using `System.currentTimeMillis()` this might cause problems with NTP synchronization and leap seconds. Alternatively, it is possible to calculate a base value of `System.nanoTime()` and calculate the real time using an updated value of `System.nanoTime()`. This paper describes the aforementioned pattern. We show that the pattern can create a statistically significant, but small performance improvement in the JVM.

1 Introduction

Many applications require exact time recording: Observability of software needs to record what is done why [2], and high frequency trading requires recording the exact time when a transaction is finished. In the JVM, the time can be obtained using `System.nanoTime()`, which is more accurate and returns relative times, and `System.currentTimeMillis()`, which is less accurate, but returns absolute times.

To obtain timing information in the JVM, the pattern we call the **Time Measurement Pattern (TMP)** occurs: The start time from `currentTimeMillis()` and the initial value of `nanotime()` are stored. Subsequently, when the time needs to be obtained, the `nanotime()` is called, the initial value of `nanotime()` is added, and the difference is added to the initial value of `currentTimeMillis()`. Consequently, `nanotime()`'s high accuracy can be used for an absolute timestamp.

In this paper, we (1) describe the pattern and its occurrences in depth, and (2) quantify its overhead for two observability frameworks, for Kieker [8] and Apache SkyWalking.¹

The remainder of this paper is structured as follows: First, we describe timers in the JVM. Subsequently, we describe the TMP and its occurrences. Afterwards, we describe its application in Apache SkyWalking. Then, we compare our work to related work. Finally, we give a summary.

¹<https://skywalking.apache.org/>

2 JVM Timers

To obtain the time, two main methods exist: `System.currentTimeMillis()` and `System.nanoTime()`.

`System.currentTimeMillis()` obtains the absolute time since 1.1.1970 using a call to `clock_gettime(CLOCK_REALTIME, ...)` and **converting** the result to milliseconds. `CLOCK_REALTIME` is regularly synchronized using NTP.² Since the quartz clock inside of the computer *drifts*, the obtained time is before or after the NTP standard time. Additionally, synchronization issues with NTP and leap seconds might lead to the clock jumping, resulting in wrong ordering of events [5, p. 289f].

`System.nanoTime()` yields the absolute time since some fixed point in the past, which might be the system start time. It uses a call to `clock_gettime(CLOCK_MONOTONIC, ...)`. It can be used to get a time span length by reading the time twice. Based on the temperature of the quartz clock inside of the computer, it might run faster or slower than the real time. The system might therefore decide to speed up or slow down the clock, but it is guaranteed that the clock never jumps. The accuracy of `CLOCK_MONOTONIC` is expected to be higher than the accuracy of `CLOCK_REALTIME`.

Since the time is retrieved often by various parts of custom applications and the operating system itself having a syscall for every instance would be very time-consuming. In modern Linux systems, the time of the default timer source *tsc* is therefore stored in the virtual ELF Dynamic shared object (vDSO).³ The vDSO is a separate area in the memory which applications from the user space can access without having to issue a syscall.

3 The Time Measurement Pattern

The TMP is visualized in Figure 1: On initialization of the tracing (`init`), the initial value of `currentTimeMillis()` and `nanotime()` are obtained. These are stored into the start time (`start`) and the anchor of the monotonic time (`delta`). Later on, when the time function `getTime` is called, `nanotime()` is called again,

²https://man7.org/linux/man-pages/man3/clock_gettime.3.html

³<https://man7.org/linux/man-pages/man7/vdso.7.html>

Domain	Framework	Class
Obs.	OpenTelemetry	AnchoredClock
Obs.	Kieker	SystemNanoTimer
Obs.	Elastic APM	EpochTickClock
HFT	Aeron-io	OffsetEpochNanoClock
HFT	Chronicle	SystemTimeProvider

Table 1: Pattern Implementations in Frameworks

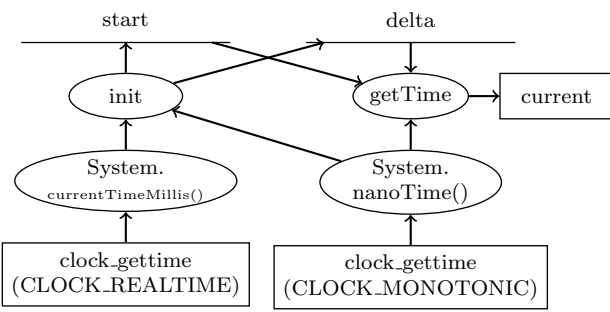


Figure 1: The Time Measurement Pattern

the amount of nanoseconds passed since the start time is calculated and added to the start time to get the **current** realtime. During one JVM run, the start time is never gathered again. Hence, there are no jumps in the obtained time values, but the time might deviate from the real time for longer running processes. The pattern can be observed in various frameworks. Table 1 shows example usages.

OpenTelemetry’s documentation assumes that the application of the TMP improves the performance: “There is also a performance improvement in avoiding referencing the system’s wall time where possible. Instead of computing a true wall time for every timestamp within a trace, we compute it once at the local root and then anchor all descendant span timestamps to this root’s timestamp.” Elastic APM on the other hand documents it as a pure accuracy improvement: “`System.currentTimeMillis()`, which uses wall clock time is not suitable for measuring durations because of the aforementioned possibility of clock drifts.”

4 Overhead Quantification

To quantify the overhead of applying the TMP, we compare the application and the bare measurement using `System.currentTimeMillis()` in two frameworks: Apache SkyWalking and Kieker. SkyWalking supports a variety of technologies, including Java, C#, Node.js, and Go. Due to this polyglot technology support, it is widely used in cloud-native projects. By default, SkyWalking obtains start and end time of spans using a direct call to `System.currentTimeMillis()`. The used code is depicted in Listing 1. Furthermore, we used the Kieker observability framework, where the timer `$T` can be specified using `-Dkieker.monitoring.timer=$T`. Kieker’s `SystemMilliTimer` and `SystemNanoTimer` use `currentTimeMillis()` and `nanoTime()`, and by default, the

values is serialized without conversion. Besides the unit of timestamps, the obtained records have the same structure. In order to measure the timer overhead directly, we also created a `ZeroTimer`, that always returns zero and thereby does not create any overhead.

Listing 1: Typical code to obtain timings

```
public boolean finish(TraceSegment owner) {
    this.endTime=System.currentTimeMillis();
    owner.archive(this);
    return true;
}
@Override
public AbstractTracingSpan start() {
    this.startTime=System.currentTimeMillis();
    return this;
}
```

To check the efficiency of our results, we used the microbenchmark MooBench [10]. MooBench measures the overhead of observability frameworks by automating the instrumentation of a system under test. The experiments were carried out on AMD Ryzen 7 5700G using Rocky Linux 9.8 with Kernel 5.14.0-611.54.1 and OpenJDK 17.0.20 and on Raspberry Pi 5, using Debian Linux 13.6 with Kernel 6.18.34+rpt-rpi-2712, using the default timer source *tsc*. All experiments were repeated with 30 JVM starts. The results are available in our dataset.⁴

Using MooBench’s default parameter, we observed the execution times depicted in Table 2. While we see a slightly worse performance for Kieker ($t = 3.30$ on Ryzen 7, $t = 4.13$ on Raspberry Pi), we don’t see a significant change in SkyWalking ($t = 0.79$ on Ryzen 7, $t = 0.92$ on Raspberry Pi). Based on the zero timer, we can estimate that the raw overhead of a monotonic timer call is around $38ns$ ($= \frac{1,451.05-694.58}{10.2}$, division by 10 due to call tree depth and 2 since there is a timer call at the beginning and the end of each traced method). This value is slightly higher than in our last study [9], where we inferred a timer overhead of 30 % of $1,383.33ns$ for Kieker, which leads to $21ns$ ($= \frac{0.30 \cdot 1,383.33}{10.2}$).

We repeated the experiments with growing call tree depth and found similar behavior: Small, but significant performance differences. Since our experiments were executed on current hard- and software, we assume that it is generalizable for similar execution environments. Under different circumstances, e.g., in very limited IoT hardware, the TMP might also yield an substantial overhead reduction.

Based on analysis of the source code, the only difference between `nanoTime()` and `currentTimeMillis()` is the conversion back and forth.⁵ Hence, the worse performance of `currentTimeMillis()` could be fixed adding an method without unit conversion in the JVM code.

⁴<https://doi.org/10.5281/zenodo.21809764>

⁵jdk: os_posix.cpp#L1387

Configuration	Mean	Std. Dev.
	/ns	/ns
Ryzen 7 5700G		
Kieker-java		
No instrumentation	64.10	0.18
Binary file (<code>s.NanoTimer</code>)	1423.70	22.07
Binary file (<code>s.MilliTimer</code>)	1451.05	39.63
Binary file (Zero timer)	694.58	59.78
SkyWalking-java-patched		
No instrumentation	64.11	0.16
Patched (MONOTONIC)	4532.24	183.76
Regular (REALTIME)	4518.29	221.67
Raspberry Pi 5		
Kieker-java		
No instrumentation	114.02	0.12
Binary file (<code>s.NanoTimer</code>)	4165.31	81.41
Binary file (<code>s.MilliTimer</code>)	4269.08	111.14
Binary file (Zero timer)	2817.61	120.22
SkyWalking-java		
No instrumentation	114.51	0.12
Patched (MONOTONIC)	10926.80	172.76
Regular (REALTIME)	10870.00	285.79

Table 2: Execution Times Using Different Timers

5 Related Work

Kupferberg et al. [1] analyze the resolution and time consumption of timers, treating them as a black box. They find that the Time Stamp Counter (TSC) has the best resolution and lowest accuracy, and that Java’s `System.nanoTime()` is also a good choice. By the approach of treating timers as a black box, they neglect effects that are happening beyond the systems they used for experimentation, e.g., that the TSC is considered to be unstable due to CPU scaling.⁶ While their work analyzes timers in general, we discuss the timer usage in tracing frameworks.

Shipilëv [4] examines similar properties: latency and granularity, with latency being time consumption in the terminology of Kupferberg et al., and granularity being the minimum time between two calls. Shipilëv finds that hardware at the time of writing had a overhead and granularity of 30ns. We found that timers take 39ns; since the current study took place ten years later, this indicates that no significant improvement in timers took place.

Gregg [6] examined the overhead of time measurement after a regression that happened due to a switch from CentOS to Ubuntu. He found that switching the time source from `xen` to `tsc` yields significant performance improvement; due to missing availability of cloud resources, we could not check whether MooBench yields similar results when using the cloud.

Merino and Otero research timers in the context of containers [7], where time inside of containers should be consistent, even if containers are re-started on different hosts. This is done by adding offsets to the

timers in the containers time namespace. They find that the overhead to timer calls is less than 4% for all scenarios.

6 Summary

We describe the Time Measurement Pattern in the JVM, a pattern that combines one call to `currentTimeMillis()` and continuous calls to `nanoTime()` to get the current timestamp with minimal variation. By using updated versions of Kieker and Apache SkyWalking, we showed that a small, but significant overhead reduction can be caused by using the TMP instead of the `System.currentTimeMillis()`.

Estimating the accuracy and overhead of time function calls remains a relevant research topic. We see two future research directions: First, we focused on the overhead inside the JVM; a language-independent examination of the accuracy could quantify to what degree `CLOCK_MONOTONIC` is more accurate than `CLOCK_REALTIME`. Second, new techniques like Google Spanner TrueTime [3], and AWS’s open-source ClockBound⁷ assure consistent time across distributed systems; especially for distributed tracing, comparing their accuracy and overhead to the local clocks would be interesting.

References

- [1] M. Kuperberg, M. Krogmann, and R. Reussner. “Timer-Meter: Quantifying Properties of Software Timers for System Analysis”. In: *QEST 2009*. 2009. DOI: 10.1109/QEST.2009.49.
- [2] B. H. Sigelman et al. *Dapper, a large-scale distributed systems tracing infrastructure*. Tech. rep. Google, Inc., 2010.
- [3] J. C. Corbett et al. “Spanner: Google’s globally distributed database”. In: *ACM Transactions on Computer Systems (TOCS)* 31.3 (2013), pp. 1–22.
- [4] A. Shipilëv. *Nanotrusting the Nanotime*. Sept. 2014. URL: <https://shipilev.net/blog/2014/nanotrusting-nanotime/>.
- [5] M. Kleppmann. *Designing data-intensive applications*. 2017.
- [6] B. Gregg. *The Speed of Time*. <https://www.brendangregg.com/blog/2021-09-26/the-speed-of-time.html>. 2021.
- [7] X. Merino and C. E. Otero. “The cost of virtualizing time in linux containers”. In: *2022 IEEE Cloud Summit*. IEEE. 2022. DOI: 10.1109/CloudSummit54781.2022.00016.
- [8] S. Yang et al. “The Kieker Observability Framework Version 2”. In: *Companion of the 16th ACM/SPEC ICPE*. 2025, pp. 11–15. DOI: 10.1145/3680256.3721972.
- [9] D. G. Reichelt et al. “Benchmarking the Overhead of Distributed Tracing Agents”. In: *Proceedings of the 17th ACM/SPEC ICPE*. 2026, pp. 147–161. DOI: 10.1145/3777884.3797004.
- [10] D. G. Reichelt et al. “MooBench: A micro-benchmark for performance overhead measurement of observability tools”. In: *Journal of Open Source Software* 11.123 (July 2026), p. 10400. DOI: 10.21105/joss.10400.

⁶linux: tsc.c#L47

⁷<https://github.com/aws/clock-bound>