ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA153

REAL-TIME JAVA TO SUPPORT THE DEVICE PROPERTY MODEL

C. Cardin, M.-A. Galilee, J.-C. Garnier, K. Krol, M. Osinski, A. Stanisz, M. Zerlauth
CERN, Geneva, Switzerland

Abstract

Today's front-end controllers, which are widely used in CERNs controls environment, feature CPUs with high clock frequencies and extensive memory storage. Their specifications are comparable to low-end servers, or even smartphones. The Java Virtual Machine (JVM) has been running on similar configurations for years now and it seems natural to evaluate the behaviour of JVMs on this environment to characterize if Firm or Soft real-time constraints can be addressed efficiently. Using Java at this low-level offers the opportunity to refactor CERNs current implementation of the device/property model and to evolve from a monolithic architecture to a promising and scalable separation of the area of concerns, where the front-end may publish raw data that other layers would decode and re-publish. This paper presents first the evaluation of Machine Protection control system requirements in terms of real-time constraints and a comparison of the respective performance of different JVMs. In a second part, we will detail the efforts towards a first prototype of a minimal RT Java supervision layer to provide access to the hardware layer.

INTRODUCTION

The device property model has been used at CERN since the exploitation of the Proton Synchrotron (PS) in the early 1960s and the subsequent Super Proton Synchrotron (SPS). In this model, users are aware about devices. A device is uniquely named, and represents a physical device such as a Beam Interlock System, or a software service such as an abstraction layer to a group of systems. Each device belongs to a Class. The Class defines the Properties that can be used to access the device. A property can support 3 types of operations: get, set, subscribe.

A set operation allows the user to send a value to the property. The device will then handle the values in the appropriate way, typically in case of a hardware device, writing it into a register. A get operation allows the user to read a value from the property. Typically in case of a hardware device, this actions reads a value from a register. A subscribe operation allows the user to receive notifications from the property. Typically reading the value from a register following the refresh frequency of the hardware device.

The device property model is supported at CERN by the Controls Middleware (CMW [1]) for the communication. The control software for any equipment consists basically in a software server exposing the devices along with their properties using CMW.

The Machine Protection interlock systems, like the Beam Interlock System (BIS [2]), are designed in a highly dependable way: The entire critical functionality is implemented in a hardware layer, while the software layer brings only monitoring and control features. If the software is unavailable, the system safety is not compromised and it will continue to be able to perform the most critical safety functions and put the machine into a fail-safe state, if required. However, controlling and e.g. re-arming the system will not be possible anymore without a fully operational software layer, hence the availability of the accelerator could be reduced.

The software layer for the interlock system is therefore not safety critical. Simply put, it provides monitoring and control operations to:

- get all the board registers
- get the history buffer
- set a register value

The clients of this API are multifold. Operation crews and interlock experts are using Graphical User Interfaces (GUIs) to control the interlock systems, to perform operations on them and to know their state. As interlock systems are a crucial component of accelerators handling large amounts of stored energies, they are also used by automated sequences [3], and by online and offline analysis tools.

The interlock software layer also handles some repetitive procedures. They send values to users that subscribed to properties or verify some states in the hardware. In addition, the software layer reacts upon asynchronous events such as a Post Mortem [4] dump request in order to send its internal states and history buffer to the Post Mortem service for diagnostic. The handling of repetitive procedures and asynchronous events as well as synchronizing the accesses to the hardware buses are the most relevant use cases for the required real-time behavior.

The Machine Protection use cases for the interlock controls supervision software is to acquire data from the hardware registers and rolling buffers at a frequency of 1 Hz. Only a few routines are executed periodically. Measurements have shown that the longest execution time was 130 ms. This leaves a large portion of the one second period to handle asynchronous requests from users and other tasks.

The Machine Protection software layers also involve a few virtual devices backed by Java servers, in order to provide a better level of abstractions to its clients, and to perform more advanced functionalities, e.g. to decode on the fly all boolean signals published by the interlock devices.

The Front-End Controllers (FEC) on which the supervision layer is executed are becoming more powerful. The current architecture has 2 cores and 1 GB of RAM, while the minimum setup of the next generation osf FEC is 4 cores and 4 GB of RAM. These ressources are more than enough to run a Java Virtual Machine. The FEC runs Real Time Linux.

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA153

The growing FEC computing capabilities, the available processing time in the period, and the experience developing with the Device Property model hardware classes backed by C++ servers and virtual classes backed by Java servers triggered the deeper investigation of a possible real-time Java implementation of the Machine Protection control software.

Real-time software means that it should be deterministic and reliable in providing an answer to a request, according to configured *deadlines*. The consequences of missing a deadline helps categorizing real-time software:

- Hard: Missing a deadline results in a total system failure. Ex: automotive
- Firm: If the deadline is overdue, the data is invalidated and thereby the quality of the service is degraded. The system integrity is not at risk. Ex: Weather forecast.
- Soft: The more the deadline is overdue, the more the data is useless. This is tolerated as long as the delay is recoverable. Ex: video streaming.

In case of the Beam Interlock System control software and other Machine Protection systems, the control layer implements a simple data access to read and write data. Actually, no use case requires hard real time constraints at the moment. Missed deadlines in the data access use case will not cause a system failure.

Most of the use cases require soft real time, except for the sending of Post Mortem data that requires firm real time. There will be a few challenges to address: the way Java manages memory, the fact that the program will need to perform native calls to use resources out of the JVM, and the use of third party libraries that were not necessarily developed having Real-Time in mind. The hypothesis is that the garbage collector operations may fit within the spare time available at each iteration, and that the overhead of native calls and third party libraries will be under control.

This contribution is based on a preliminary research [5] performed in 2015. It presents in a first section the benchmarking tool. The second section presents an overview of the Java environment and the confirmation of the initial benchmarking results. The third and fourth sections then respectively present the real-time performance of native calls and message loggers. The fifth section summarizes the evaluation of the BIS real-time procedures implemented in Java.

BENCHMARKING TOOL

The evaluations were performed based on a synthetic workload which was designed to be as close as possible to reality. It however means, that more accurate metrics must be collected from a real test-bed later on.

The benchmarking software runs many iterations of a single task. A task corresponds to a synthetic use case from the original study, or to a use case involving native calls, or logging, or the execution of BIS real time procedures. The software was designed to avoid having benchmarking code removed during the optimization of the JVM at runtime. A benchmarking exercise starts with a warm-up phase that ensures that all the code paths are used and that all the classes are properly loaded, so that the measurements are not performed while the JVM is still spending time on initializing its runtime environment.

The collected evaluation metrics are the following:

- period: the time between each task iteration.
- deadline: the time-limit for a task to be considered successfully executed.
- response time: delay between the execution request and the response of the system.
- iteration: basically the load of the system, each iteration doubles the load of the previous iteration.

A required deadline and period are provided as parameters to the benchmark, as well as the number of iterations and the time to spend on an iteration. Benchmarks were executed on a front-end controller with 2 cores and 1 GB RAM, and on a desktop machine with 8 cores and 16 GB RAM.

The same benchmarking tool has been used for all the evaluations that follow, the difference being the operation under study.

JAVA VIRTUAL MACHINES AND GARBAGE COLLECTORS

Based on our experience developing supervision layers in C++, we consider that two kinds of objects will live in the Java Virtual Machine of our supervision layer: short life-cycle objects that will be created in order to handle communication requests, e.g. domain objects that encapsulate the data to be sent to properties getters or subscribers, or domain objects that encapsulate the data received from properties setters, or instances related to a single execution of a real-time procedure. These objects will be stored in the Eden Memory [6] and will be collected from there, never making it to a longer term memory storage.

Long life-cycle objects of the supervision software are service components that will never be collected: typically a scheduling service, an API controller, etc. There are also long-term domain objects and data structures that are typically used by the real-time monitoring layer. All these objects will end up in the Old Generation memory and will actually be active throughout the entire lifetime of the running JVM.

Garbage Collector operations on the Eden space are called Minor GC and are usually very light and fast. Operations on the old generation storage is more ressource intensive. The Java Garbage Collector operation is famous for its *Stop the World Event* where all the threads are stopped until a garbage collection operation is completed. In a Real-Time Java context, these events should be avoided, or minimized. Depending on the garbage collector implementation, *Stop the World Events* are more or less frequent and triggered under certain conditions.

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA153

The original study analyzed the Hotspot [7] JVM from Oracle. This JVM provides different garbage collector implementations. The original study focused on Garbage-First (G1) [8]. Parallel (or Throughput) GC [9] has been used as a reference as it is the default GC for Hotspot. Surprisingly, it was also the solution that showed the most interesting results. Both are included in this study. Shenandoah [10] was added in our study as it promises to be an ultra-low pause time garbage collector, trying to collect the garbage in parallel to the program execution. Concurrent Mark Sweep (CMS) GC [11] was omitted in the original study and was not included in this one, as it is very comparable to G1 and more likely to perform extensive pauses in the program execution. In addition to Hotspot, other JVMs were studied in the original study: Azul Zing [12] and JamaicaVM [13]. In this evaluation only Zing has been considered, because JamaicaVM did not address properly some of our requirements and its results were already poor in the original study. Zing's garbage collector algorithm is almost pause free for the running program.

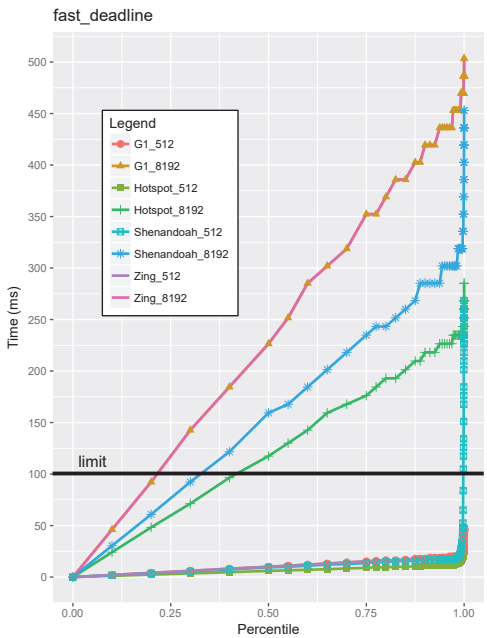

Figure 1: Performance of intensive memory operations of different JVMs and GCs regarding the deadline. Zing and G1 are completely overlapping on the graph. Discussions with the Zing development team helped us clarifying that Zing's garbage collector and the G1 are actually very similar.

Figure 1 shows that Hotspot with Throughput GC is performing surprisingly well, much better than the other combinations for the iteration 8192, though the deadline limit is missed after the 0.4 percentile. At iteration 512, all solutions were roughly equal, except Shenandoah which goes beyond the deadline limit. If the response time is considered, Shenandoah becomes more interesting, performing much better than the other solutions at iteration 8192.

The main observation is that the deadline and response times are acceptable, with the upper boundary being 160 ms. Considering memory operations it seems that any solution could be a match for our use case, with Throughtput GC being the most interesting at the moment, keeping an eye on G1 and Shenandoah that are still under development and may improve in the future. It was clear however that more tests had to be performed on the real use cases of the interlock supervision.

NATIVE CALLS

The supervision program running in the JVM must access with read and write operations to the hardware devices. This is typically performed by using a standard CERN library that encapsulates calls to the actual driver which takes the form of a Linux kernel module. It is therefore necessary for a Java program that wants to access the hardware devices to perform native calls. Native calls are also necessary for the supervision software to use the CERN accelerator timing [14]. The timing library is a crucial component of the supervision: it provides access to local and central timing information. The supervision software needs this to know the accurate time of the system, or to learn about timing events like the start of a new accelerator cycle or a beam extraction. This library is only provided in C/C++, therefore native calls are necessary. An important feature is that the supervision software must be able to listen to certain events in order to start certain procedures.

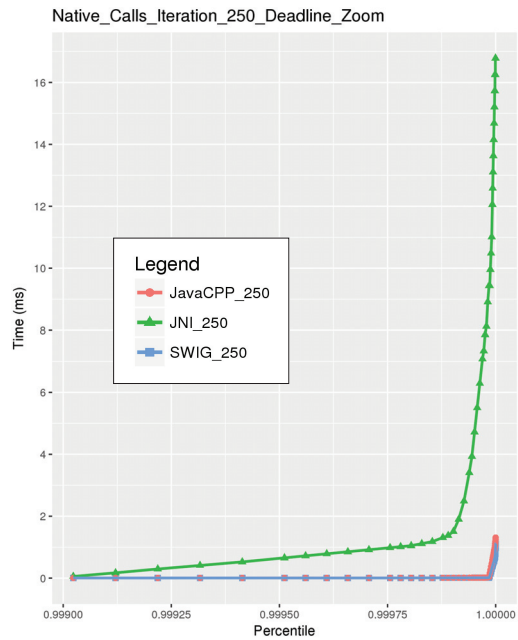

Figure 2: The evaluation of the deadline for the native calls at iteration 250.

Some solution focus on performance, such as JNI [15], SWIG [16] and JavaCPP [17], whereas other solutions make the implementation easier, this is for example the case of JNA [18]. JNA was omitted from the study while the other three were considered. The tests for the native call libraries relied on reading and writing a variable in a C memory space. Figure 2 shows that JavaCPP and SWIG are roughly

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA153

equivalent. Our preference went to JavaCPP because of its ease of use.

MESSAGE LOGGERS

A message logger is a key component of the software supervision. It is very important to record and persist properly "who did what and when" on a system. It means any action from a user on a property must be recorded: subscribing, un-subscribing, getting or setting. In case of a set, the data a user sent needs to be recorded, in order to know what was the user's intent on the system. Depending on the level of diagnostic information required during certain periods of operation, multiple logs could be written. This must not impact the real-time behavior of the software supervision. There are various ways to send logs remotely. The research focused on the well spread and documented libraries logback and log4j2. The focus was given to remote logging, as the front-end controllers do not have local drives where to store logs. Logging to syslog-ng via UDP [19] and TCP [20], and pushing directly to logstash [21] via TCP were studied. The aim is to persist the log messages in an Elasticsearch [22] instance. All log messages that are emitted must be properly persisted and indexed, so that investigations can be performed when an issue occurs. The benchmarking of logging technologies consisted in writing a certain number of logs to a different support.

Figure 3 illustrates that logback seems the most suitable to log a large number of messages to a remote service, outperforming the other implementations by far. It was verified that no messages were dropped, an unwanted behavior that could have explained this good performance.

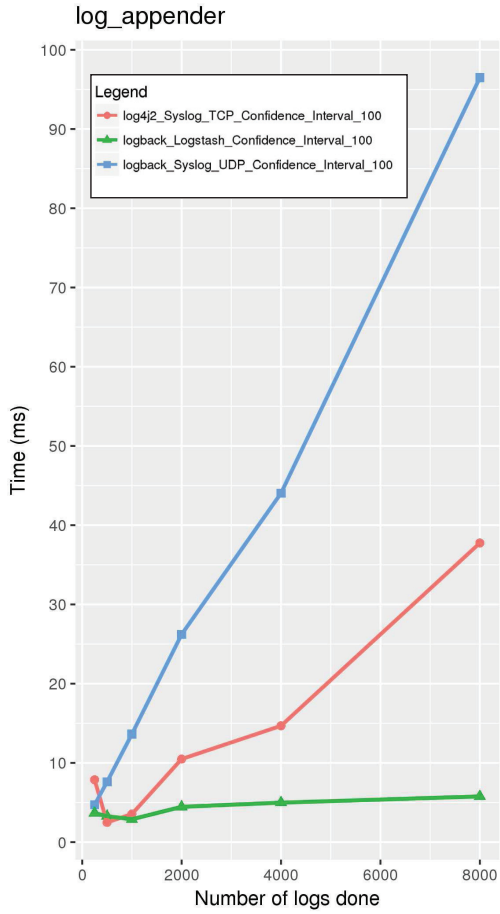

Figure 3: The performance of log4j2 and logback logging remotely a parameterized number of messages.

REAL-TIME PROCEDURES FOR THE BEAM INTERLOCK SYSTEM

The last and most important step was to study the currently existing real-time procedures of the BIS. As they only existed in C++, they had to be rewritten in Java. This review was the opportunity to bring a few corrections and optimizations to their behavior. The way to measure the execution time was not exactly the same for Java and C++. We had all the control necessary for the Java benchmarking as we were reusing the benchmark tool used for the other evaluations. The C++ measurements are coming from the real control software running on real hardware devices. Therefore the following comparison shown in Table 1 is not entirely representative of a Java/C++ comparison, but it gives some perspective about the Java implementations. In addition, the Java Post Mortem procedures are not comparable with their C++ counter part as they were entirely refactored to improve their behavior. One can see that the performance of the Java procedures is acceptable and this fosters further studies of the use of real-time Java for hardware supervision.

OUTLOOK

The benchmarking results are very encouraging regarding a Java implementation of the BIS supervision software. No blockeing issues were encountered, but on the contrary it seems that all operations can be performed in firm and soft real-time within the 1 second period the BIS software supervision is following.

It was therefore decided to implement a first proof of concept of a simple Java supervision for BIS hardware devices. The Spring framework [23] has been used for two main reasons: its dependency injection mechanism and its scheduler. Once the JVM and the garbage collectors had been evaluated for real-time performance, the aim was to produce a proof of concept of a complete supervision software for the BIS. The dependency injection is a very convenient feature to decouple the user code from a potential general framework. It improves the testability of the code, and features like component scan ease the integration of user code in the framework instance: adding a real time procedure can consist in just creating one new class. The scheduler is obviously responsible for triggering the recurrent real-time procedures.

The minimal Java framework is therefore able to trigger real-time procedures based on timing events originating from the Central Timing, schedule recurrent real-time procedures at 1 Hz, and provide control properties via CMW. The program has been running on a FEC in a testbed for several weeks. The next steps are to:

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA153

Table 1: Evaluation of the Real-Time Procedures for the BIS Supervision

Procedure	Mean Execution Time (ms)	Standard Deviation (ms)	Confidence Interval 99% (ms)	C++ Mean Execution Time (ms)
Update register values	6.65	2.65	[6.63, 6.68]	25
Notify Logging Service	3.64	1.70	[3.61, 3.64]	25
Post Mortem Acquisition	10.84	2.90	[10.81, 10.87]	N/A
Post Mortem Dump	142	19.0	[140, 143]	N/A
Timing Correction	0.230	0.759	[0.169, 0.292]	130
Perform Software Permit	0.195	0.762	[0.168, 0.223]	0.150
Write Register Value	0.187	0.798	[0.158, 0.216]	0.150

- Perform evaluations with a real load, with real hardware devices and an environment similar to the accelerator operation to provide meaningful metrics.
- Monitor the garbage collector performance accurately, and study the lifetime of the generated objects thoroughly.
- Follow up on the latest developments of the Java environment, e.g. the Java 9 release on 21st September 2017.

In case hard real-time constraints appear in the future, non-standard solutions like Javolution [24] could be considered.

ACKNOWLEDGMENTS

The authors would like to thank Jens Egholm Pedersen for the valuable work he performed when carrying out his bachelor thesis “Predictable firm real-time Java”. His work served as a basis to further explore the feasibility of real-time Java.

REFERENCES

[1] A. Dworak *et al.*, “The new CERN Controls Middleware”, *CHEP2012*, New York, USA (2012).

[2] B. Puccio *et al.*, “The CERN Beam Interlock System: Principle and Operational Experience”, *IPAC’10*, Kyoto, Japan (2010).

[3] R. Alemany-Fernandez *et al.*, “The LHC Sequencer”, *ICALEPCS2011*, Grenoble, France (2011).

[4] O. Andreassen *et al.*, The LHC Post Mortem Analysis Framework, *ICALEPCS2009*, Kobe, Japan (2009).

[5] J. Pedersen, “Predictable firm real-time Java”, unpublished.

[6] http://www.oracle.com/technetwork/articles/javase/index-jsp-136373.html

[7] http://www.oracle.com/technetwork/articles/javase/index-jsp-136373.html

[8] https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/g1_gc.html

[9] https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/parallel.html

[10] https://wiki.openjdk.java.net/display/shenandoah/Main

[11] https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/cms.html

[12] https://www.azul.com/products/zing

[13] https://www.aicas.com/cms/en/JamaicaVM

[14] J. Lewis *et al.*, “The CERN LHC Central Timing, A Vertical Slice”, *ICALEPCS07*, Knoxville, Tennessee, USA (2007).

[15] http://docs.oracle.com/javase/8/docs/technotes/guides/jni

[16] http://www.swig.org

[17] https://github.com/bytedeco/javacpp

[18] https://github.com/java-native-access/jna

[19] https://tools.ietf.org/html/rfc768

[20] W. Richard Stevens, *TCP/IP illustrated, Vol. 1: the protocols*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1993.

[21] https://www.elastic.co/products/logstash

[22] https://www.elastic.co/products/elasticsearch

[23] https://projects.spring.io/spring-framework

[24] javolution.org