TOWARDS HIGH PERFORMANCE PROCESSING IN MODERN JAVA BASED CONTROL SYSTEMS

M. Misiowiec, W. Buczak, M. Buttner, CERN, Geneva, Switzerland

Abstract

CERN controls software is often developed on Java foundation. Some systems carry out a combination of data, network and processor intensive tasks within strict time limits. Hence, there is a demand for high performing, quasi real time solutions. Extensive prototyping of the new CERN monitoring and alarm software required us to address such expectations. The system must handle dozens of thousands of data samples every second, along its three tiers, applying complex computations throughout. To accomplish the goal, a deep understanding of multithreading, memory management and interprocess communication was required. There are unexpected traps hidden behind an excessive use of 64 bit memory or severe impact on the processing flow of modern garbage collectors. Tuning JVM configuration significantly affects the execution of the code. Even more important is the amount of threads and the data structures used between them. Accurately dividing work into independent tasks might boost system performance. Thorough profiling with dedicated tools helped understand the bottlenecks and choose algorithmically optimal solutions. Different virtual machines were tested, in a variety of setups and garbage collection options. The overall work provided for discovering actual hard limits of the whole setup. We present this process of architecting a challenging system in view of the characteristics and limitations of the contemporary Java runtime environment.

OVERVIEW

Diagnostics and Monitoring at CERN

Device monitoring and alarm management systems have long been running separately as part of the CERN accelerator controls environment [1]. Both deal with a vast quantity of diverse devices scattered around CERN installations to acquire, process and redistribute their data. What makes a difference is the meaning of data. Monitoring system periodically collects the statuses of the supervised infrastructure. In principle, data shows no irregularities, as the underlying equipment is by and large healthy. The alarm system, however, carries the information already known to have indicated abnormal conditions. They must be swiftly delivered, through the Operators, to the experts capable of solving the problem. In one case both systems clearly intersect: when the monitoring data flags an erroneous situation and must be converted into an alarm.

Prototyping

Recently it has been decided to prepare and evaluate an architecture of the combined solution, a system that would coalesce all the types of data, devices and users, embracing both worlds. Significant effort has been put to model several possible outcomes based on well established CERN software packages [2][3]. To constitute a solid foundation for such analysis and discover genuine physical boundaries we would certainly face, an independent prototype was constructed [4].

For prototyping it was decided to only cover the most critical topics. The outcome was focused and tuned to maximal extent to satisfy the hardest requirements. From a technical viewpoint such effort helped us to:

- determine if the requirements can be realistically matched with available components,
- establish hard limits for performance & scalability,
- find optimal solutions for the most resource-consuming operations.

In this paper we discuss selected subjects - design choices, issues, best practices - that have become important in achieving expected quality.

ARCHITECTURE

Problem Description

CERN monitoring system works in a heterogeneous environment. It has to communicate with a variety of *devices*, through a set of different protocols. Many devices are regular CERN Linux servers. Majority, however, typically front-ends to the physical equipment, operate on less standard or legacy architectures, unknown to Java products. Devices provide monitoring or alarm data in form of *metrics*, sample values describing their state. Metrics are published either periodically (*monitoring agents*) or immediately after an erroneous condition is discovered (*alarm agents*). Devices and the network that connects them are *not fully reliable. No real-time* infrastructure is used to communicate with them. Once delivered to the monitoring server, metrics have to be verified, correlated, converted into alarms, etc. Such an individual *calculation* based on metric(s) is defined in a *rule*. Decisions are made which of them and in what form should be afterwards delivered to the clients - the GUI applications that present the state of the CERN infrastructure to the Operators. Information flow must be *preserved* for future inquiries too.

Requirements

The system is expected to fulfil a long list of requirements, of which a subset relevant to this paper is presented. For this paper we focus on the performance, scalability and reliability issues. The system must be at the time of writing able to:

- acquire alarm metrics from around 10000 devices,
- deliver alarms within bounded time frame,
- acquire 50 metrics/min from 2500 monitoring agents,

- in total, on average, handle 2000-5000 metrics/sec,
- in total, on average, perform around 5000-20000 rule calculations/sec,
- sustain traffic peaks in form of alarm avalanches with several times bigger throughput.

Considering constant expansion of the CERN infrastructure, the amount of equipment is bound to increase. Thus, our system would hopefully accommodate for double the numbers presented above.

Among the points omitted here are: two-way communication with devices, managing misbehaving devices, reconnections, filtering the input, failover mechanism or any configuration aspect.

Design

The overall design follows a classic multilayer architecture with clear separation of concerns. The bottom tier is represented by scattered devices - data providers. Above lies a data acquisition tier (DAQ), which caters for a variety of middleware protocols to communicate with the devices. Those include JMS, Yami4 [5], SNMP and CERN extension of CORBA called CMW [6]. DAQ is located on a separate machine due to the nature of its intensive self-contained computations. Another machine hosts the middle-tier whose main component is a rule engine (RE), a core calculation unit of the system. All the components above the device layer are built in Java.

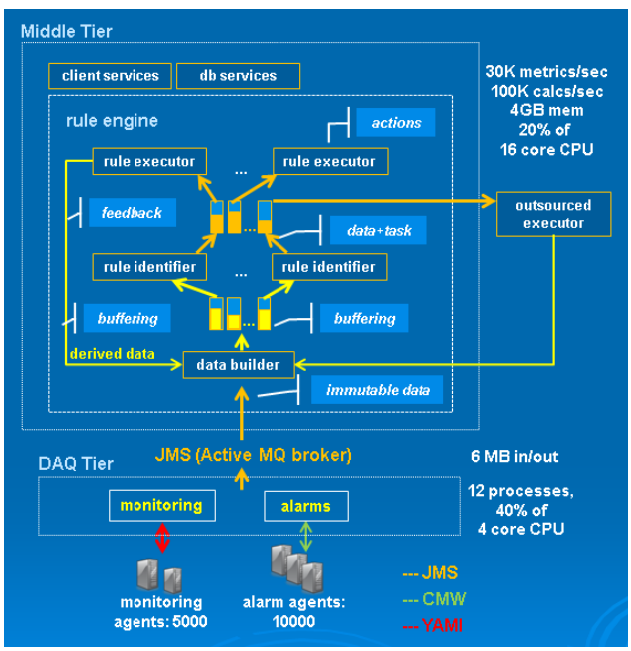

Figure 1: System architecture.

Figure 1 gives a system overview. The numbers associated show the final setup and the results of enduring tests performed with the ultimate architecture. Bottom tier consists of various classes of devices supervised by their DAQ counterparts using middleware protocols.

Now we decompose the presented architecture to look thoroughly at the issues solved throughout and major topics that had arisen.

TECHNICAL PATH

Uniformity

Our bottom tier is a heterogeneous environment. The messages holding data samples are of different shape, the middleware they flow through is divergent. What we pursue however is the opposite: a unification of resources.

CERN control systems largely rely on JAPC library [6]. JAPC operates with different middleware protocols while preserving a common high-level API. It adopts the concept of a *parameter,* which describes a device and a class of data it offers. For each parameter JAPC is able to establish an appropriate transmission channel to its underlying device. JAPC unifies data acquisition across the whole bottom tier. It seamlessly consolidates the access to over 10000 parameters.

The data itself is consistently shaped thanks to introducing a common *metric* structure. It encapsulates every type of input data. JAPC monitoring threads translate the incoming traffic into a stream of unified metrics, whereby acting as pipeline data converters. Such metrics are complete: they hold all the information required for further processing. From then on they are marked immutable, which is crucial to efficient data distribution and task parallelism of the middle-tier.

DAQ outgoing stream of metrics is again divided into new parameters. Such parameters represent logical domains rather than individual devices (alarms vs. monitoring data, computers vs. equipment, etc). It renders an amalgam of almost randomly acquired source data into a narrowly classified stream of metrics. Moreover, DAQ can decide on the pace those metrics are fed into the output JMS channel (recurrent vs. continuous flow) and the size of messages holding them. The latter prevents JMS environment from being flooded with miniscule pieces of data. It reduces the total overhead lied upon the messages too. In any case, the JMS infrastructure must be tuned accordingly to system needs, considering message persistency, lifetime, etc.

Data unification settles a foundation for effective parallelization in the middle-tier. It notably improves the testing capabilities of the whole system. Presented mechanism partly obliterates the need for distributed caching, an attractive yet highly problematic solution.

Decomposing for Concurrency

Rule engine is the core processing unit of the middle-tier server. A *rule* is a definition of calculation performed on the set of metric values, typically the most up-to-date ones. An example is a threshold rule that verifies if a metric value is within defined boundaries (e.g. temperature) or a mathematical formula where several disparate metric values need to be used to calculate the result. For sake of clarity, we skip all more advanced topics on rule calculations here. The output of the rule can be any action performed by the server, e.g. throwing an alarm, but also an *artificial* metric (created by the server, not monitored devices), which is in turn fed into another rule. Hence, rules form a tree structure which metrics

flow through. The same metric can be obviously used by many rules. As rules are laid out at the startup, it is always known where every original or artificial metric should be passed to.

Rule engine is a classic example of task, data and pipeline decomposition [7]. Each metric is an independent immutable piece of data. Rule is a task, which must be performed every time a set of arguments, metrics, is available. The input metric channel, along with the paths they flow between rules, constitutes a data-driven pipeline processor, a chain of responsibility. The overall structure vows for concurrent architecture.

There are three tiers of processing in that model. The first is an input channel, where immutable metrics are prepared and made available. The second tier is made up of rule *resolvers* (*identifiers*). They place each metric among the arguments of every rule they belong to. Once the argument list is complete, a self-contained *rule job* representing a single rule calculation is created. The last tier is comprised of rule *processors,* which take such jobs and perform associated calculations, likely producing artificial metrics (Fig. 1). As metrics and rule jobs are immutable and independent, numerous rule identifiers and rule processors can act autonomously. It gives the ground for parallelism.

The crucial is however what separates those three tiers. Java 6 Queue implementations (FIFO buffers) constitute the buffering layer [8]. The first layer holds metrics, the second keeps rule jobs, however they are both identically set up. Buffering layer implements the asynchronous Reader/Writer pattern. What has a decisive impact on the performance of the rule engine is the number of buffers, their sizes and the strategy for choosing one to read or write. Long tests proved the following to perform optimally in our setup. The layer is composed of a few buffers (2-4), each has a capacity of about half a million elements. Writers and readers are at least twice as numerous as the buffers. The strategy on reading the data is straightforward: pick the buffer holding the least elements. For writing, use the smallest buffer at the time for a period, e.g. 1000 writings, then repeat choosing. Both round-robin or random choosing strategies appeared to have fallen short of the selected policy. Its efficiency depends on many aspects, some being low level hardware properties (e.g., CPU cache hits).

Having a buffering layer configurable, including various R/W policies, is a precious feature. It greatly helps to find an optimal setup for a given architecture. During the tests it is essential to monitor the contents of the buffers and watch against any missed writings.

Because of the hierarchical composition of rules there is a natural tendency to break calculations into periodic stages, *rounds*. It partly roots in the fear that generating artificial metrics and passing them further may spark off an avalanche of computations. On the contrary, we have found no legitimate reason for staging the processing. The rule engine is left to run at its natural speed continuously. Artificial metrics are not directly handed to the following rules, but fed back into the input channel. It gives a perfectly consistent view on both processing and data. It also results in the highest throughput of all, reaching *100000* non-trivial calculations a second sustained for over a week without a single metric lost.

It is worth mentioning the thread scheduling discipline is typically OS-specific. Therefore a designer must guarantee that multicore CPU will be properly utilized for a given setup. It might need additional tweaking of the OS parameters.

Garbage Collection

Once the architecture is optimized at the level of data structures, algorithms and task parallelization, tuning the garbage collection remains as the pivotal job. Understanding how garbage collectors affect an application is crucial to laying out realistic boundaries on how performing the system can be.

The combination of extensive computing (non-trivial method calls) and frequent memory allocations contributes to numerous garbage collections within a time frame. Moreover, in such case garbage collectors require to operate on those heap structures (e.g. old generation) that inflict stop-the-world pauses - all the application threads get inhibited for a considerable period [9]. Unlike the Java Real Time System, none of four main garbage collectors available in Standard Edition guarantees non-interruptible work for the underlying application. They however differ in the lifecycle of such interruptions and the use of threads to perform on.

Although our system in general exhibits only soft real time requirements, in order to achieve massive continuous processing of metrics it is essential to impose almost hard real time requirements on the JVM. Testing various configurations of GC options for heightened number of calculations, we experienced a common pattern - failing scenario. Once the GC starts pausing the application on a regular basis, it will inevitably break down having less and less chance to make up for the lost time.

Having thoroughly tested our prototype with JRockit, Parallel-Compacting, Concurrent Mark-Sweep and GarbageFirst (G1) collectors in a variety of options, we found the last two establish the most sustainable environment. Due to the nature of processing and uniqueness of our setup such a statement cannot be generalized. It remains only as an incentive for every designer to never neglect the topic.

Table 1: Example Options Used with Tested GC's

GC	Selected options
PC	UseParallelOldGC, DisableExplicitGC, ParallelGCThread, CompileThreshold, MaxGCPauseMillis, NewSize
CMS	CMSIncrementalMode, CMSIncrementalPacing, CMSIncrementalDutyCycleMin, CMSIncrementalDutyCycle
G1	GCPauseIntervalMillis, G1ParallelRSetUpdatingEnabled, G1ParallelRSetScanningEnabled

GarbageFirst is a server-style collector, targeted for multi-processors with large memories. It also well utilizes the 64 bit architecture. G1 is considered the most efficient mechanism available in Java Standard Edition and has been decided a default solution shipped in JDK7.

Garbage collectors undergo constant development, changing their shape across JVM versions. Modern research concepts, as *split bytecode verification* or *lock coarsening,* enhance the overall process.

Memory Allocation

Organising memory allocation is a subject complementary to garbage collection. Java developer is offered to set up at least the boundaries of memory available to the application through a series of JVM options. Besides, the choice of the computer architecture and operating systems notably influences the outcome. As 64 bit multicore servers have become a prevailing standard, excessive use of the heap space emerged tempting. When the 4GB limit per JVM is surpassed (effectively much less depending on the operating system, see Table 2) an application can handle massive volumes of data in memory. The number of threads allowed to be created is also substantially larger. Nonetheless, experimenting with the settings shows it immediately brings a performance penalty in form of extended garbage collections, a hindrance we struggle to overcome at any cost. The other known effect of upgrading to 64 bit architecture comes with the need to operate on extended memory addresses. It is generally considered to moderately hamper the speed of processing.

Table 2: Memory Available in 32 bit Java SE [10]

OS	Heap Size
Linux	2.2-3 GB
Solaris	3.5-4 GB
Windows	1.4-1.8 GB
MacOS	<4GB

As for the garbage collection, choosing the right computer architecture and finally tuning the memory available affects the overall quality of our application. All those topics should be consciously scrutinized.

Profiling

Testing a complex Java system cannot be effectively carried out without the help of profiling tools. A rudimentary support for code orchestration comes with a JDK tool, VisualVM [11]. It allows to watch a lifecycle of the application with respect to its CPU, memory and thread consumption. Monitoring these characteristics is indispensable when tuning the garbage collection and memory allocation properties.

VisualVM and JConsole rely on the JMX support that JVM instance offers. We can easily profit from the feature by exposing some parts of our system as JMX calls. It lets manage the system during tests using a standard out of the box tool.

CONCLUSIONS

Java software design should always be considered with respect to the characteristics and shortcomings of the modern Java runtime environment. Conversely, garbage collection and memory allocation settings are often perceived superfluous, concise choice of data structures and concurrent routines considered secondary. Performance and scalability tests tend to be underestimated.

CERN monitoring and alarm system is a high throughput, semi real-time and most importantly reliable application based on the Java components. Architecting its prototype has brought tangible evidence how some often neglected issues determine the overall result of software development.

We have shortly discussed a selection of topics whose deep understanding may be vital to achieving a quality distributed software. They are more than likely to influence an outcome of designing any Java-based system that shares similar requirements.

REFERENCES

[1] M. Buttner et al., “Diagnostic and Monitoring CERN Accelerator Controls Infrastructure: Diamon Project First Deployment in Operation”, Proceedings of ICALEPCS’2009, Kobe, Japan.
[2] Technical Infrastructure Monitoring, timweb.cern.ch
[3] S. Deghaye et al., “CERN Proton Synchrotron Complex High-Level Controls Renovation”, Proceedings of ICALEPCS’2009, Kobe, Japan.
[4] W. Buczak and M. Misiowiec, “Diamon/Laser prototypes – report”, CERN EDMS, 2011
[5] M. Sobczak, “Programming Distributed Systems with YAMI4”, Lulu, 2010
[6] V. Baggiolini et al., “JAPC - Java API for Parameter Control”, Proceedings of ICALEPCS’2005, Geneva, Switzerland
[7] T. Mattson and B. Sanders and B. Massingill, “Patterns for Parallel Programming”, Addison-Wesley Professional, 2005
[8] B. Goetz and T. Peierls and J. Bloch and J. Bowbeer and D. Holmes and D. Lea, “Java Concurrency in Practice”, Addison-Wesley Professional, 2006
[9] E. Bruno and G. Bollella, “Real-time Java programming with Java RTS”, Sun Microsystems, 2009
[10] Java forum topics, http://stackoverflow.com
[11] VisualVM, http://visualvm.java.net