Analyzing Real-Time Java: Deadline Experiments and Comparison with C

Fernando G. Tinetti[1], Demian KÍč, Fernando L. Romero
III-LIDI, Fac. de Informática
Universidad Nacional de La Plata
[1]Also with Comisión de Investigaciones Científicas Prov. de Bs. As.
La Plata, Argentina

***Abstract*—We have designed and implemented a set of experiments in order to compare a Real-Time Java (RT Java) virtual machine implementation with *plain* Real Time Linux (i.e. Linux + rt-preempt patch). Given the characteristics of real-time execution and applications, we consider the specific code and/or benchmarks as important as the runtime and environment configurations for performance evaluation (as well as for production environments, as a matter of fact). Experiment results are compared, for obtaining RT Java overhead over real-time Linux with plain C programming. Besides usual hardware and basic real-time Linux configurations, we explain some specific details of RT Java that should be taken into account in all RT Java implementations, such as the garbage collector and its impact on meeting time constraints. We experiment with Real Time Java and compare results with similar or the same experiments using the C language, the *de facto* standard in real-time computing. Even when different languages are used, the main metric is the one used in the real-time field: missed/met deadlines. We mostly used standard experimentation programs and developed a specific one for having fair timing comparison among experiments.**

Keywords—Real-Time Java, Real-Time Systems, IoT, Hard and Soft Real-Time Deadlines

I. Introduction

Real-time systems and applications are being analyzed since decades ago [6] [7] [17] [19] [30]. Today, many of the systems and applications formerly considered in the real-time area are now considered part of classical mission-critical applications [6] [7], Internet of Things (IoT) [12] [31], embedded systems [9] [19] [17] and other areas. Even when each area has its own specific details and requirements, all have several common issues beyond its relationship with real-time processing. Among those common issues, the software development languages, libraries, and environment should be considered from the software productivity point of view. High level object oriented languages such as Java are considered the most appropriate from the point of view of software production [25] [8]. However, those high-level language and libraries are complex to analyze and implement for real-time processing [23] [6] [30].

While the well-known undebatable baseline for real-time processing is at the real-time operating system (RTOS) level, the software development process is still a problem. Once the operating system is able to provide the minimum facilities for real-time computing/determinism such as the Linux rt-preempt patch [11] [24], several aspects of software development have to be considered at the same time, and most of them depend on each other. Java is considered a high level, stable, and popular programming language, and efforts have been made to take advantage of those characteristics in many different applications [18] [29] [4]. Also, there has been a sustained effort for developing a Real-Time Specification for Java (also known as RTSJ), which initially produced the current stable 1.0.2 version [28] [5] and the 2.0 version is currently being defined [15]. Besides having a current stable specification, it is possible to use several implementations, one of which is the so-called JamaicaVM [1].

Our main work is focused on defining a set of programs and environment scenarios for analyzing periodic tasks deadline/s. We use as many well-known programs/benchmarks as possible, and define a new one only if necessary. Our scenarios set several parameters such as real-time threads, priorities, and free/overloaded system so that it is possible to have a broad (while manageable) number of tests and possible runtime environments. We also focus our work mostly in hard real-time applications, since they have the most constrained requirements in terms of real-time deadlines (i.e. a failure in meeting the deadline is considered catastrophic). Even when focused in hard real-time tasks, we are able to take into account soft real-time tasks. Soft real-time tasks are those for which missing a deadline is not considered as catastrophic, but the system response is losing quality/usefulness as the elapsed time is longer than that defined as the required by the application. Once again, a predictable computing system/task is preferred over a fast computing system/task in the context of real-time computing, so minimum jitters are expected in the elapsed time taken by the system as a reaction to an external event and for processing periodic tasks.

Even when real-time systems are well defined, it is not always possible to take into account all possible scenarios a priori. Thus, specific programs/benchmarks are defined and used in order to have a better understanding of system behavior

in different scenarios. We start including in this paper those we consider useful in the context of RT Java specification and implementation, so that it is possible to be compared with a plain Linux with rt-patch programmed in C. As an initial approach, we have also taken some measurements in non real-time systems just to have experimental data of the timing constraints and system performance.

RT Java includes several requirements for implementations that make it useful for real-time systems. Among the most useful definitions we could identify [28] [5]:

- Fixed priority scheduling (with preemption).
- Memory definitions and garbage collector scheduling: garbage collector is preempted if necessary and has more information about memory used by objects.
- Real time threads and asynchronous event handlers, so that the scheduler has precise information about activation times (e.g. periodic tasks).
- Priority inversion handling/avoidance via an abstract class with subclasses defined for different priority inversion avoidance algorithms.

And we should take advantage of/configure all of them in our experiments. It is worth mentioning that all the RT Java definitions are useful only if running in top of a real-time operating system, otherwise it is not possible to maintain runtime predictability.

II. Related Work

Most of the work in the real time computing area has been developed at the embedded and/or operating system level, as reflected in current bibliography [6] [7] [9] [11] [12] [17] [19] [30] [31]. Most of those books/publications are currently used in real-time computing and/or programming courses at University graduate and post-graduate levels. Java real-time programming can be considered as starting, even when there is registered work on Java Specification Requests since about two decades ago [27]. Most of the problems are due to the "gap" between the specification, which usually imposes specific requirements, and actual implementations, which must successfully meet those requirements. And the real-time field has quantitative requirements not always simple to satisfy.

Having well defined Real-Time Specifications for Java as its current 2.0 version [15], it has been important to implement and evaluate real-time applications. Some of the effort is reflected in currently referenced books, such as [5] [6]. Most of the work is devoted to software engineering, such as those mentioned above as well as [14] [21] [22]. Furthermore, several previous publications are directly focused on specific implementations, such as [10] [26] [13].

Our proposal is to provide several experiments, measurements, and settings/scenarios for advancing and/or getting new insight over those previously published works while being independent of specific products/implementations. Since the real-time experiments are well-known in the real-time Linux environment, we will do our best effort to have similar experiment/s in the real-time Java environment.

III. Experiments: Code and Environment

Clearly, the underlying execution environment must be a real-time one. We have chosen Linux with rt-patch because it is a real-time operating system, and it is widely known and available for a large number of computing platforms. Real-time computing in this environment is straightforward, since it handles/directly implements the most necessary facilities such as threads, priorities, preemption, etc. [11]. The standard and most useful benchmark is used in this environment: cyclictest [20], along with also well-known auxiliary programs: taskset (for handling process/threads affinity) and stress (for computer workload, so that the system is fully loaded while using the benchmark) [16]. Thus, Fig. 1 schematically shows the experiment environment in real-time Linux.

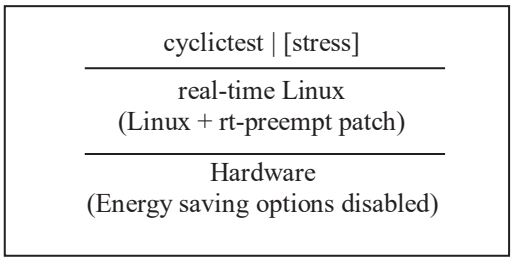

Figure 1: Real-Time Linux Environment.

We have chosen JamaicaVM as the real-time Java experiment execution environment because: a) it implements Java, b) it also implements a significant fraction of the RTSJ as well as some RTSJ extensions, such as a real-time garbage collector [2], and c) it is available for free in its Personal Edition version. The JamaicaVM documentation clearly describes the limitations of the RTSJ implementation as well as specific JamaicaVM implementation details beyond the RTSJ. Thus, portability issues are easily identified and can be solved in advance, and the JamaicaVM environment also includes a switch for controlling strict RTSJ compliance. The basic benchmark used in the JamaicaVM is similar to cyclictest, called jittertest [3]. However, we have developed one more benchmark srtp (from "simple real-time period") in order to measure the elapsed time since the activation of a periodic real-time thread and the beginning of its real execution. Neither RTSJ nor JamaicaVM provide a way to measure the "waiting time" since a thread gets the scheduler ready queue (activation time) and it effectively begins to use the CPU (CPU execution runtime). Actually, this simple and specific benchmark was developed for obtaining the same experimental data as that provided by cyclictest. The key point of the srtp is starting at a specific time and, given that periodic tasks should be activated at specific times (defined by a static parameter), the latency is computed as shown in Eq. (1), where

- *rst*: real start time, taken by the thread itself as the experiment advances.
- *cdt*: cyclic determined time, i.e. the time at which the thread should have started is execution (using the CPU).

$$y = rst - cdt \quad (1)$$

Fig. 2 schematically shows the experiment environment in a real-time Java virtual machine running in a real-time Linux.

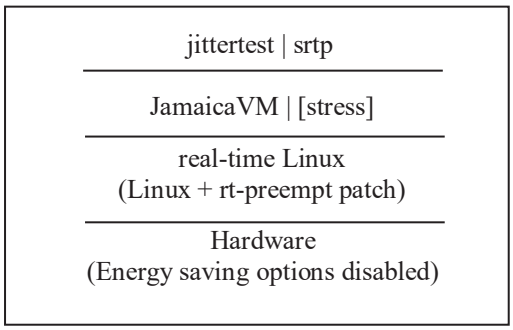

Figure 2: Java Real-Time Environment.

All experiments are carried out in a relatively current hardware computer: AMD A10-6800K APU (4 CPUs) @4120 MHz, 16 GB de RAM, Linux Ubuntu 14.04 LTS 64 bits, vanilla kernel 3.14.39 (which is patched later with the rt-preempt patch). The rt-patched kernel has been compiled with the following settings: a) disable dynamic ticks, b) disable latency tracing, c) disable energy saving options (except those from the ACPI module), d) enable high resolution timers, and fully preemptable kernel. Also, some BIOS have been changed: a) disable CPU throttling, b) set clock to "extreme mode" (max. frequency). Finally, the Linux kernel startup option idle=poll was set to avoid further interference of energy saving kernel features. Unfortunately, there are multiple energy saving options set at different levels (kernel compile, computer BIOS, and kernel load parameter). Furthermore, those specifically unrelated with the Linux kernel (i.e. at the BIOS) may depend on the specific hardware (motherboard) provider.

IV. Experiments: Results

Initially, some preliminary results will be shown, with measurements taken over short-periods of time (a few seconds/ minutes). Fig. 3 shows different cyclic test short-time experiment scenarios, some of them as a quantitative reference of possible problems meeting deadlines in non real-time environments.

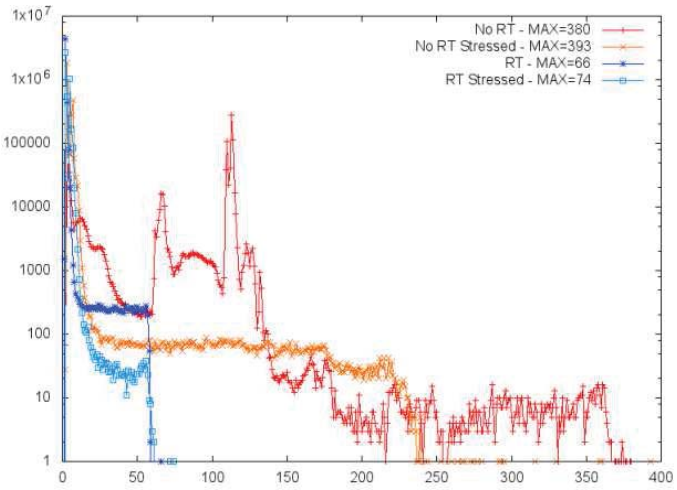

Figure 3: Preliminary cyclictest Results.

In Fig. 3, the measured delay times are shown on the x-axis and the number of occurrences are shown on the y-axis, with an interval of 500 µsec:

- No RT: a standard Linux operating system without rt-preempt patch. As expected, it has the worst results, with a maximum delay of 380 µsec.
- No RT Stressed: adding workload to the standard Linux system so that no CPU is idle during the experiment. As expected, the result in terms of maximum delay is worse than the same scenario without stress: 393 µsec.
- RT: a real-time Linux without stress. The maximum delay is drastically reduced to 66 µsec.
- RT Stressed: adding stress to a real-time Linux implies a degradation of the maximum delay of about 12%: 74 µsec.

Fig. 4 shows the jittertest results with stress (JamaicaVM running on real-time Linux). The period was set to 150 µsec and the maximum measured period was 271.13 µsec. Period measurements are computed as the times between two releases [3], which is not what we are interested in, since we want to compare delays with the cyclictest benchmark. Therefore, we are going to use our own srtp benchmark thus avoiding indirect comparison/s.

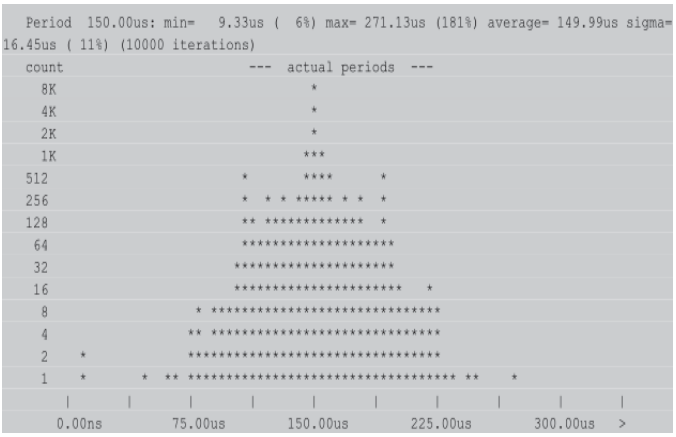

Figure 4: Stressed System jittertest Results.

After having the previous set of preliminary experiments, we have decided to use the cyclictest benchmark for 24-hours experiments and use our own srtp benchmark for direct comparison of results. The cyclictest is used on real-time Linux and srtp is used on a JamaicaVM running as a real-time process in the same real-time Linux. In both scenarios, the stress utility is used for overloading the system with many low priority processes/threads.

Fig. 5 shows the cyclictest results over 24 hours in a mostly idle (non-stressed) real-time Linux. The maximum delay time was measured as 38µsec. The cyclictest results over 24 hours in a stressed (i.e. running the stress utility so that every CPU has some thread available to run) real-time Linux are shown in Fig. 6. In this scenario, the maximum delay time was measured as 46µsec, i.e. about 21% worse than that for the non-stressed environment.

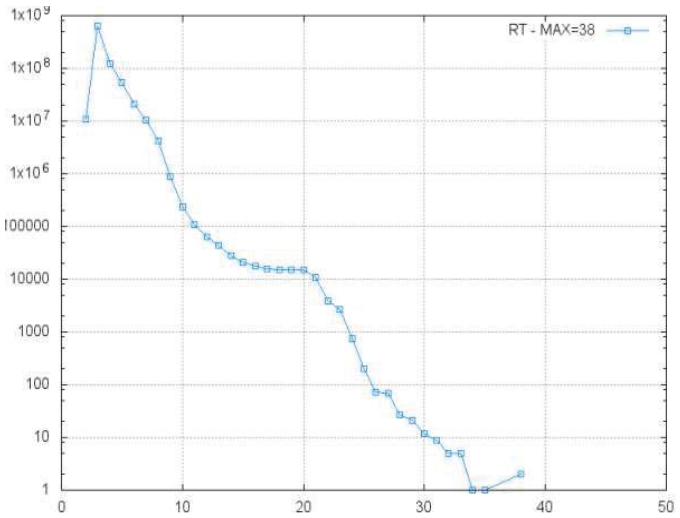

Figure 5: 24-Hours cyclictest Results without Stress.

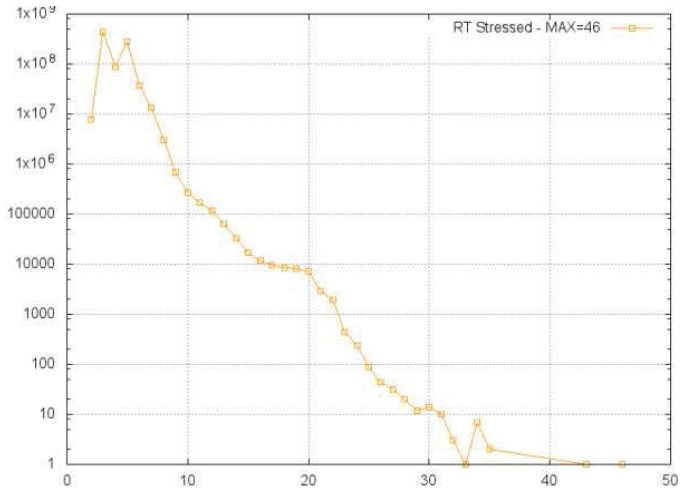

Figure 6: 24-Hours cyclictest Results with Stress.

All the long-term (24-hours) experiments show that most delays are below 20µsec. Take into account that the y-axis in Fig. 4 and Fig. 5 above is in logarithmic scale. The 24-hour stressed experiment is particularly useful. Based on those results, we would be able to determine that the system handles hard real-time processing tasks with deadlines not minor than 50µsec.

Fig. 7 shows the experiment measurements using srtp on a JamaicaVM running in a real-time Linux with stress, with a maximum delay of less than 150µsec. The y-axis in Fig. 7 is the % of delay measurements taken in µsec shown in the x-axis. Experiment measurements are almost the same for the non-stressed runtime environment. Clearly, most of the delays are between 50 and 70 µsec, but the system would not able to handle hard real-time tasks with deadline below 150µsec. The overhead/greater delay time measured in the JamaicaVM is clear: only a few thread delays were below 50µsec, while in the real-time Linux all the measurements made with cyclictest are clearly below 50µsec. The srtp benchmark allows direct timing comparison, so the difference is not due to noise generated by indirect timing derivations.

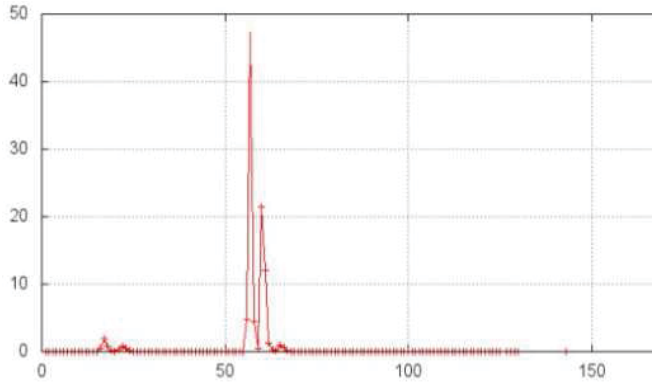

Figure 7: 24-Hours srtp Results with Stress.

Even when the real-time Java implementation effectively has worse delays than those in the real-time Linux, there is a strong indication of real-time performance. Even under stress (by overloading the system with a specific program, so that it is not possible that a CPU is idle) the delay measurements are highly concentrated in the interval 50-70µsec, only a few delays are greater than the upper bound of that interval, and all delays are below 150µsec. Therefore, it is possible to identify 150µsec as a lower bound for hard real-time computing in this environment.

It is worth noting that the lower bounds for hard real-time processing identified by cyclictest and srtp in the different environments depend at least on hardware details. However, both benchmarks provide a methodological way of having the specific bounds in the specific hardware to be used.

V. Conclusions and Further Work

We have measured experimental real-time bounds in a specific Real-Time Specification for Java implementation (JamaicaVM). We are able to compare performance with the "native" real-time Linux because we have implemented a specific benchmark program. Actual experiments have shown that

- RTSJ implementation performance provides a strong environment for real-time computing, since there are not unbound delays in long-term experiments under heavy workloads.
- RTSJ implementation performance is about three times worse than native real-time Linux performance in terms of bounded delay time.

We expect to improve our experiments in terms of having different evaluated hardware. Thus, it will be possible to analyze the relative performance differences (whether they are similar or change depending on the underlying hardware). Also, it should be possible to develop a specific benchmark for real-time Linux so that it will be possible to directly compare jittertest-like performance values obtained in an application developed in C for real-time Linux.

REFERENCES

[1] Aicas GmbH and aicas incorporated, Aicas JamaicaVM, https://www.aicas.com/cms/en/JamaicaVM, (Feb. 2016).

[2] Aicas GmbH, JamaicaVM 8.0 — User Manual: Java Technology for Critical Embedded Systems, 2016.

[3] Aicas GmbH and aicas incorporated, Benchmarks | aicas.com, Jitter test, https://www.aicas.com/cms/en/benchmarks, (Feb. 2016).

[4] B. Bruegge, A. H. Dutoit, Object-Oriented Software Engineering Using UML, Patterns, and Java, 3rd Edition, Pearson, 2009, ISBN-10: 0136061257.

[5] E. J. Bruno, G. Bollella, Real-Time Java Programming: With Java RTS, Prentice Hall, 2009, ISBN-10: 0137142986.

[6] A. Burns, A. Wellings, Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX, 4. Addison-Wesley, 2009.

[7] G. C. Buttazzo, Hard Real-Time Computing Systems: Predictable Scheduling Algorithms, and Applications, 3. Springer, 2011.

[8] Bruce Eckel, Thinking in Java, 4. Pearson, 2006.

[9] X. Fan, Real-Time Embedded Systems: Design Principles and Engineering Practices, Newnes, Feb. 2015.

[10] S. C. Foley, Developing with real-time Java, Part 1: Exploit real-time Java's unique features, http://www.ibm.com/developerworks/ java/library/j-devrtj1/index.html

[11] L Fu, R. Schwebel. RT PREEMPT HOWTO, https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO

[12] B. Graham, M. Weinstein, The RTOS as the Engine Powering the Internet of Things, WindRiver White Paper, Wind River Systems, Inc ., Feb. 2014, available at http://www.intel.com/content/www/us/en/ internet-of-things/white-papers/ real-time-operating-system-for-iot.html

[13] A. Hall, A. Stevens, Developing with real-time Java, Part 3: Write, validate, and analyze a real-time Java application, http://www.ibm.com/developerworks/java/library/j-rtjdev3/index.html

[14] M. T. Higuera-Toledano, A. J. Wellings, Distributed, Embedded and Real-time Java Systems, Springer; 2012, ISBN-10: 1493900390.

[15] Java Community Process (JCP), SR-000282 Real-Time Specification for Java 2.0 Draft 12, Early Draft Review 2, 2015 http://download.oracle.com/otndocs/jcp/rtsj-2_0-edr2-spec/index.html

[16] A. Kili, How to Impose High CPU Load and Stress Test on Linux Using 'Stress-ng' Tool, Tecmint: Linux Howtos, Tutorials & Guides, 2015, http://www.tecmint.com/linux-cpu-load-stress-test-with-stress-ng-tool/

[17] H. Kopetz, Real-Time Systems: Design Principles for Distributed Embedded, Applications, 2. Springer, 2011.

[18] J. Lewis, W. Loftus, Java Software Solutions, 8th Ed., Pearson, 2014, ISBN-10: 013359495.

[19] Q. Li, C. Yao, Real-Time Concepts for Embedded Systems, 1. CRC Press, 2003.

[20] Linux Foundation Wiki, Cyclictest, https://wiki.linuxfoundation. org/realtime/documentation/howto/howto_rt_tools_cyclictest

[21] K. Nilsen, Developing Real-Time Software with Java SE APIs: Part 1, 2014, http://www.oracle.com/technetwork/articles/java/ nilsen-realtime-pt1-2264405.html

[22] K. Nilsen, Developing Real-Time Software with Java SE APIs: Part 2, 2014, http://www.oracle.com/technetwork/articles/java/ nilsen-realtime-pt2-2264409.html

[23] Scott Oaks, Java Performance: The Definitive Guide, 1. O'Reilly, 2014.

[24] Open Source Automation Development Lab, OSADL Project: Realtime Linux, https://www.osadl.org/Realtime-Linux.projects-realtime-linux.0.html

[25] Herbert Schildt, The Java Complete Reference, 9. Oracle Press, 2014.

[26] M. Stoodley, C. Gracie, Developing with real-time Java, Part 2: Improve service quality, 2009, http://www.ibm.com/developerworks/ java/library/j-devrtj2/index.html

[27] Sun Microsystems, Inc. The Java Community Process Program, 1998, https://jcp.org/aboutJava/communityprocess/java_community_process.html

[28] The Real Time for Java Expert Group, Real Time Specification for Java Version 1.0.2, 2006, http://www.rtsj.org/specjavadoc/book_index.html

[29] J. Visser, S. Rigal, R. van der Leek, P. van Eck, G. Wijnholds, Building Maintainable Software, Java Edition: Ten Guidelines for Future-Proof Code, O'Reilly Media, 2016, ISBN-10: 1491953527.

[30] A. Wellings, Concurrent and Real-Time Programming in Java, 1. John Wiley & Sons, 2004.

[31] L. Wood, "Real-time computing: Gateway to the Internet of Things?", Computerworld, Aug 2015, available at http://www.computerworld.com/article/2973986/high-performance-computing/real-time-computing-gateway-to-the-internet-of-things.html