

RTAI Linux based avionics sensors monitoring system

Universidad de Granada y Universidad de Napoles Federico II

Master en Ingeniería Informática

Daniele Fazzari

Contents

1	introduction	1
1.1	Real-Time Application Interface	2
1.1.1	RTAI modular structure	2
1.1.2	Run RTAI-patched kernel	3
2	Software system overview	4
2.1	Software specification	4
2.2	Software design and implementation	4
3	Real-time voter subsystem feasibility	8
3.1	Voter schedulability analysis	8
3.1.1	Computing WCETs	10
3.1.2	Computing Task-set scheduling feasibility	11
4	Project execution guide	12
4.1	VM installation	12
4.2	Compiling and executing	12
4.3	Expected behaviour	12

Chapter 1

introduction

Whether it is civil or military aviation, modern aircraft's flight instruments are equipped with **electronics systems** which enable accurate flight information feedback for aircraft's ease of control and safety, thus they play an important role in modern aviation. This electronic systems manage the information coming from the **on-board sensors**, that are measuring various parameters needed in *monitoring* and *navigation control* instruments. These sensors include: tachometers, engine parameters sensors, fuel and oil quantity sensors, pressure gauges, Altimeters, airspeed-measurement meters, vertical speed indicators and others more.. [6]

So aircraft computer systems, after processing the data coming from sensors (doing for example a *voting* operation on redundant measurements), transmit that computed information to the corresponding actuators and displays in the **cockpit** (image 1.1) to supply the pilots with additional signals in order to help him to take proper actions and precautions, and thus also preventing any kind of disaster or accident.



Figure 1.1: Boeing 757-300 Cockpit

In order to meet aircraft safety requirements, these electronic systems are **real-time systems**, thus the required tasks must be performed within a prescribed dead-line (to meet time determinism/timeliness), because otherwise failure may result in several consequences (like death of passengers). However, along with **hard real-time tasks** that have an hard real-time deadline which overrun lead to the system failure, there are also tasks with soft dead-lines which overrun just cause degradation of QoS [7] (for example a monitoring buddy task that just shows some non-critical flight informations into the cockpit).

1.1 Real-Time Application Interface

A **real time system** can be defined as a “system capable of guaranteeing timing requirements of the processes under its control”. Typically a real time system represents the computer controlling system that manages and coordinates the activities of a controlled system, which interaction is bidirectional through **sensors** and **actuators** and characterized by timing correctness constraints. Also it is desirable that time-critical and non time-critical activities coexist in a real time system [3].

To execute the real-time software system made for this project, it has been used **Real-time Application Interface** (RTAI, logo shown in figure 1.2), that is an Open-Source real-time patch for the Linux kernel, made by the aerospace engineering department of *Politecnico di Milano*, born with the purpose of making Linux **fully preemptable**.



Figure 1.2: RTAI Logo

More precisely RTAI modifies the linux kernel in the way it handles the interrupts (behaving as an **”interrupt dispatcher”**) and in the scheduling mechanism. For example it makes the **clear interrupt** operation to be redirected to an RTAI function, thus making it **”soft”** and avoiding linux drivers disabling interrupts. Thus RTAI is a real-time platform but with all the functionalities of the Linux environment (display and shell, TCP/IP, file system etc...).

1.1.1 RTAI modular structure

RTAI is composed of a bunch of modules that are added to the Linux kernel modules (infact the Unix-like Linux kernel is a monolithic, multitasking and modular operating system kernel, as shown in figure 1.3).

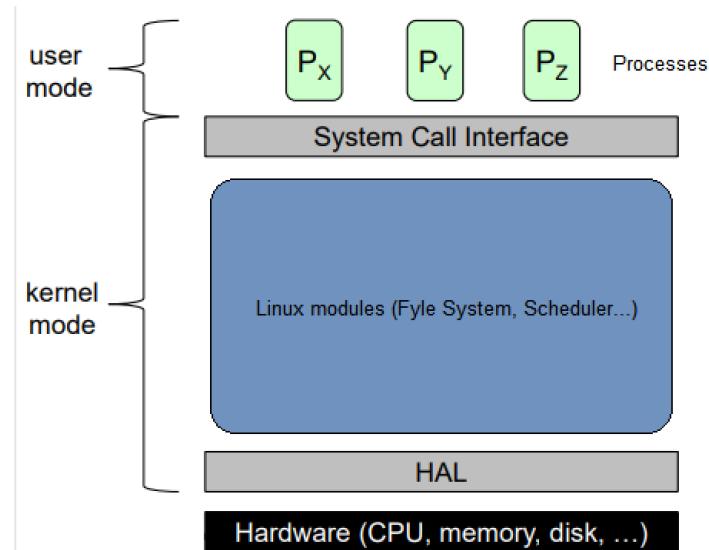


Figure 1.3: Linux modular kernel structure

Beside adding Real-time modules, the RTAI patch modifies the Linux Kernel’s Hardware Abstraction Layer (HAL), making it a **Real-Time Abstraction Layer** (RT-HAL, as shown in figure 1.4) that replaces all the original hardware interrupt handling functions. Note that RTAI gives the possibility of running **User-space RT tasks** and **Kernel-space RT tasks**.

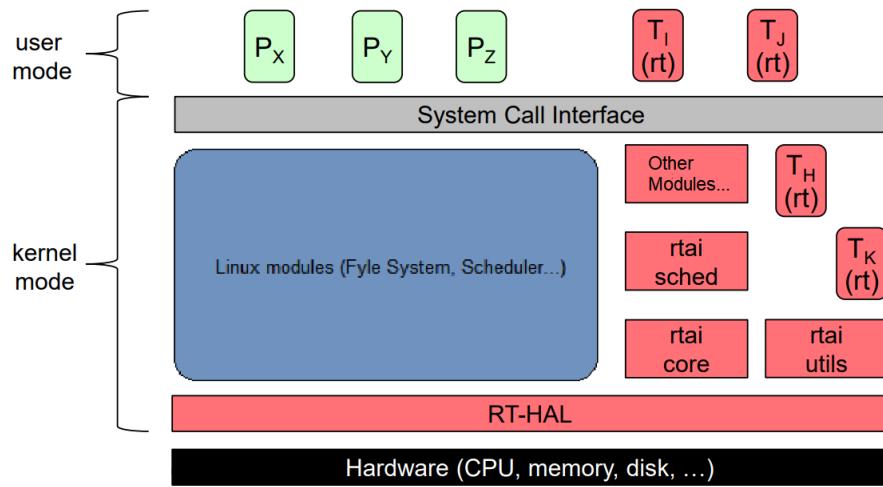


Figure 1.4: RTAI-patched Linux modular kernel structure

It's important to say that RTAI¹ patches as well other linux kernel data structures and functions, in order to support the developing and the execution of real-time application. Of particular importance and worth to mention are:

- **Task Data Structures** (or *process descriptors*), extending them with additional and useful RT information (such as task state, period, scheduling policy, priority...), in order to give the possibility to develop and run soft and hard real-time processes/threads.
- **Timing Functions, semaphore Functions, mailbox Functions, messages functions, scheduling functions** (for both user-space Linux soft real-time tasks and kernel-space RTAI hard real-time tasks) properly made for handling and support real-time code execution.

1.1.2 Run RTAI-patched kernel

Once patched the OS, when started you have the possibility to run it using the linux distribution standard kernel (in case of figure 1.5 selecting "Ubuntu" kernel option) or using another installed kernel. Selecting "advanced options" (in case of figure 1.5 "*Opzioni avanzate per ubuntu*") you have the possibility to choose to run the OS with the RTAI-patched kernel (As shown in figure 1.6).

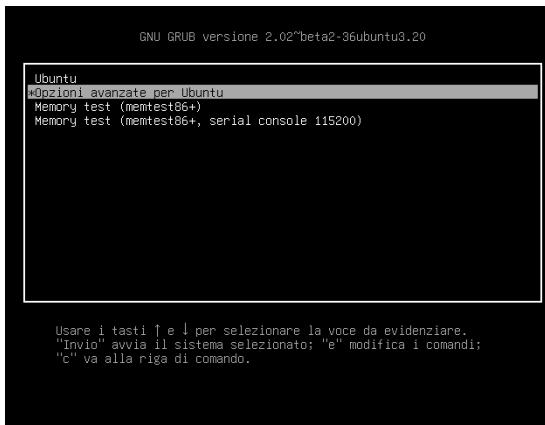


Figure 1.5

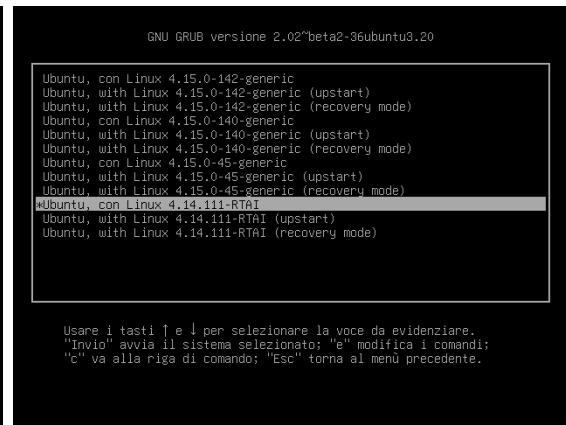


Figure 1.6

¹More information about RTAI can be found on the **RTAI Official Website** [5]. In particular you can check the last releases of the **RTAI user manual** (the one used for this project is the rev0.3 of the documentation for RTAI v3.4 [2]). More information about RTAI programming API and platform can be found in the **RTAI Programming Guide** (the one used for this project is the 1.0 [4])

Chapter 2

Software system overview

2.1 Software specification

We want to code an **avionics sensors monitoring real-time software system**. It should be capable of:

- with a **”sensor subsystem”** periodically simulate *sensors redundant measurements*, so that for each sensor type we have M redundant measurements in order to raise the reliability thus NxM periodical measurement in total (if N is the number of different sensor types), with fixed parametrizable sampling frequency for each sensor type.
- with a **”voter subsystem”** compute periodically a *voted* measurement using the correct sampling frequency for each sensor type. This subsystem has to implement the voting algorithm to produce a processed sensor measurement after receiving in input redundant sensor measurements. Note that each sensor type’s redundant measurements can be processed using a specific voting algorithm, according on the characteristics of the sensor itself.

Beside that, the system should also print to console periodically the correct processed measurements, with fixed parametric monitoring frequency. Thus the system has both soft and hard real-time requirements (we suppose the console printing not to be an hard real time task). Note that we want the voter subsystem to be executed as an hard real-time periodical process (or hard real-time thread/group of threads, depending on the implementation), infact its execution must be deterministic in terms of deadlines. However, in case of overrun, the system should trace it and count the number of overall overruns during the execution and printing it in a readable log.

2.2 Software design and implementation

I thought about making the software both **multithreading** and **multiprocessor**. Let’s see why:

1. *Why Multithreading?* Multithreading it’s mainly used to improve performance and **resource utilization**.
2. *Why Multicore in avionics?* Multicore processors provide significant **computational capacity** within a **restricted size**, **weight**, and **power**. As such, they are seen by many people as a key enabler for a wealth of new computationally intensive *safety-critical embedded systems*, such as autonomous aircraft and vehicles. However, safety-critical systems must be certified before being deployed, and *certification procedures* applicable to multicore-equipped systems are currently lacking. The key technical challenge that arises when using a multicore processor in a safety-critical setting is that of dealing with any **interference** that can occur when tasks executing

on different cores access shared hardware resources such as *caches*, *buses*, and *memory* (Such interference can cause spikes in task execution times, resulting in timing-constraint violations) [10]

Sensor subsystem ("sensor.c")

As shown in figure 2.1, the sensor subsystem¹ is the one that produces the **redundant measurement arrays** for each sensor type. More precisely it behaves like it is both the *environment* and the sensor's measurements reading subsystem, infact after producing the measurement values it writes them into a **shared memory** location (shared with the voter subsystem). This module throws N soft real-time threads, one for each sensor type. Each thread simulate and produce the redundant measurements for the corrispective sensor type. It is also made of a *monitoring buddy task* that prints periodically (with an independent fixed rate set by a parametric code macro defined in an header file) the results of the voting. as shown in figure 2.2 and 2.3 the sensor threads (thus as well the buddy task) are executing on **CPU 0**.

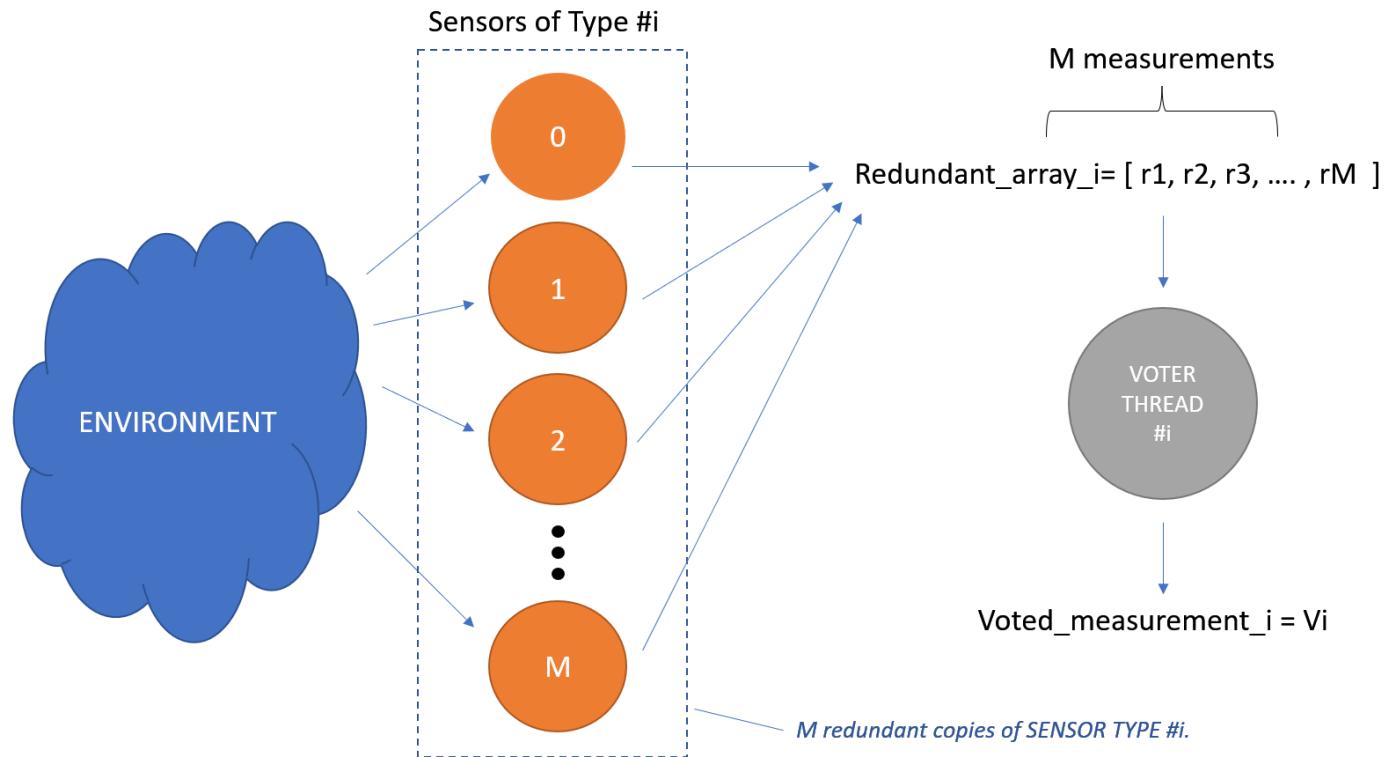


Figure 2.1: Sensor behaviour overview

Voter subsystem ("voter.c"):

As shown in figure 2.2 and 2.3, the voter subsystem ("voter.c" module²) is run on the CPU 1 (the 2nd core). It is made out of N threads, each one is associated with a sensor type. Each thread is **periodically computing** the **voted value** out of the corresponding redundant array of measurement (that is loaded by the corrispective sensor thread in the shared memory location) and thus is a **periodical RT thread** with the same period of the corresponding sensor thread (the one that generates values for sensors of type "x").

¹ "sensor.c" source code file is full with useful comments in order to better understand it

² "voter.c" source code file is full with useful comments in order to better understand it

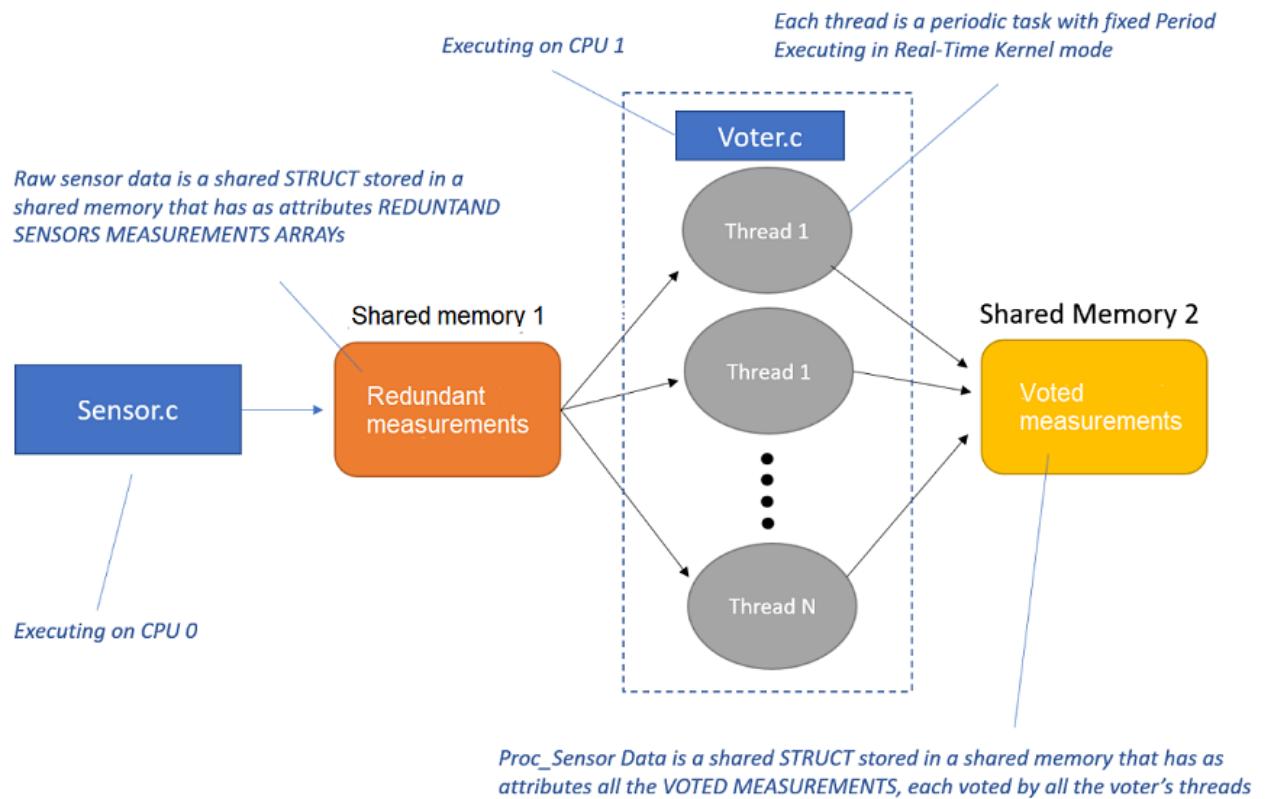


Figure 2.2: Sensor and Voter interaction scheme

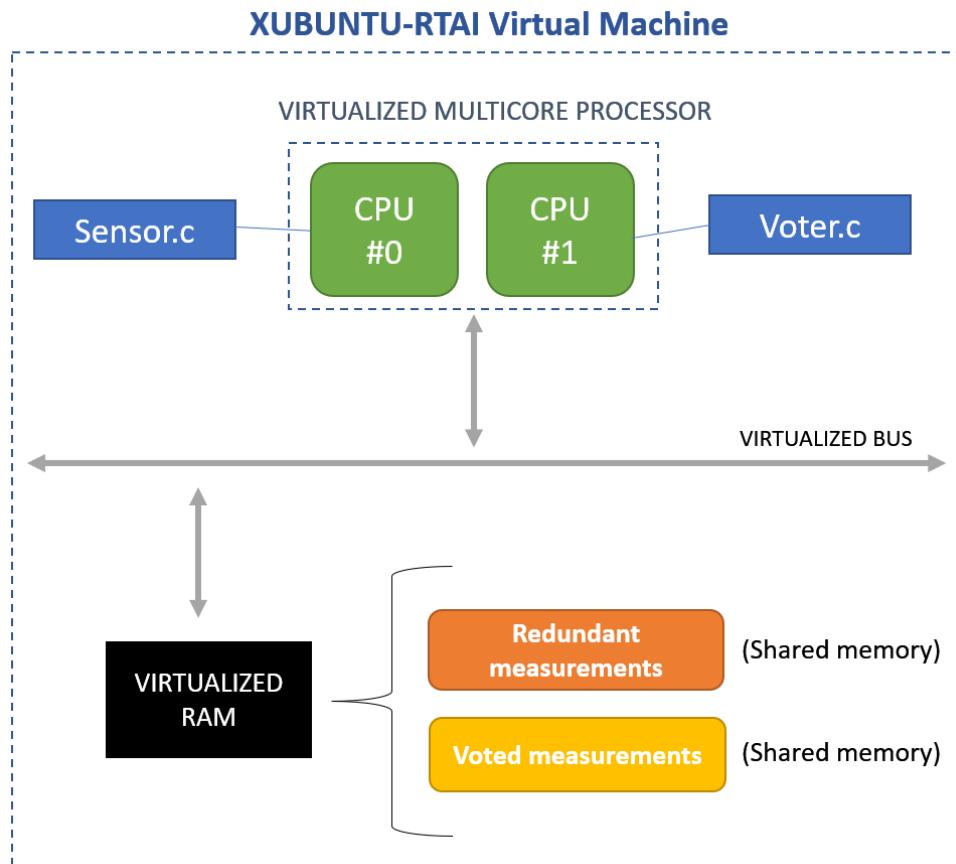


Figure 2.3: Virtual Machine CPU configuration Overview

It is important to note that the voter subsystem will be an **RTAI kernel module**, so it's threads are executed in kernel space as hard real time threads. So the voter threads have to respect their deadlines, that are established every period (as the end of the period itself). However, if the parameters of the system (such as thread's period, thread's execution times etc..) are not well set, there can be the possibility of overrun and that means that the task set executing on CPU 1 (thus where the voter threads are executing) is not feasible.

- Note that, in order to code and run this software system, RTAI has been installed on a virtual machine executing in an **Oracle VirtualBox Hypervisor** (More information about Oracle VM VirtualBox Hypervisor on the official website [9]). Virtual machines can be useful for testing and simulating real-time systems. However, it should not be used in production: the reason for this is that the performance will be less predictable. If you run the virtual machine on top of a non-real-time operating system (like in my case, VirtualBox is a *type 2 hypervisor* running in **Windows 11** environment), the host operating system could decide to do some work on its own (potentially unbounded in time), stealing resources/execution time from the guest machine. However we can still compute, with a limited accuracy level, the worst case execution times and thus analyze the feasibility of the safety-critical software system, maybe by lowering the desired ideal performances (if the software execution is feasible while executing RTAI on a VM, for sure it will be feasible while executing in a bare-metal RTAI).

Chapter 3

Real-time voter subsystem feasibility

3.1 Voter schedulability analysis

As we said, the voter subsystem is a C module that is compiled in a **RTAI Kernel Object** because it is made out of hard real-time periodical activities (in particular periodical hard RT threads). Note that in my specific implementation i've considered a task-set made of just 2 hard RT threads:

- a voter thread for sensors of type 1 (that execute "*sens1_fun*" code) with period of 20ms. In particular this function uses the "**Voter_noise**" function that can vote the correct measurement from a redundant array of integer measurements that can be affected by **noise** and with at most one broken sensor measurement (the specific noise and broken measurement implementation are commented in the sensor module).
- a voter thread for sensors of type 2 (that execute "*sens2_fun*" code) with period of 60ms. In particular this function uses the "**M_out_N_voter_bitwise**" function that can vote the correct measurement from a redundant array of unsigned integer measurements, that can be affected by at most one broken sensor measurement (the specific broken measurement implementation is commented in the sensor module), using a **bitwise** voting technique (implementation commented in the voter functions header file).

The task-set that execute these activities (the voting activities) has to be **feasible**, in order to be sure that the periodical scheduling can be made without task to be in overrun (in other words, with all the task completing its activity before the end of the i-th period). In order to compute the feasibility of the task-set's scheduling, we need to:

1. Know the **scheduling policy** used to schedule the task-set. In my case, using a specific RTAI system call ("*rt_spv_RMS()*"), the voter periodic threads (that are executing on CPU 1) are scheduled using **Rate Monotonic**¹: that makes possible to do a schedulability analysis by using the **Utilization Upper Bound Theorem**. Note that In 1973, *Liu and Layland* showed that RM is optimal among all fixed-priority assignments in the sense that no other fixed-priority algorithms can schedule a task set that cannot be scheduled by RM and also derived the least upper bound of the processor utilization factor for a generic set of n periodic tasks [1].

¹**Rate-Monotonic** is a priority assignment algorithm used in real-time operating systems (RTOS). The static priorities are assigned according to the period duration of the task, so a shorter period duration results in a higher task priority. Since periods are constant, RM is a fixed-priority assignment: a priority P_i is assigned to the task before execution and does not change over time [11]. These operating systems are generally preemptive and have deterministic guarantees with regard to response times, thus rate monotonic analysis is used in conjunction with those systems to provide scheduling guarantees for a particular application [1]

2. Compute, for every single voter thred, the **Worst Case Execution Time** (WCET). In order to do that, i used a **statistical approach**, infact it's necessary to do so because, as previously said, RTAI it's running on a Virtual Machine and that means that the latencies are not completely deterministic and thus the timeline analysis can not be deterministic.

In particular i used the notion of **confidence interval** for the mean of the execution time population to compute an "avarage WCET". For each thread i found an interval in which is contained, with a fixed **probability**, the mean execution time μ so that the WCET will be (with the same probability) the **upper bound** of the interval. In order to derive it we should need to know the poulation **standard deviation** σ , however the population has a infinite cardinality and we can not take an infinite sample of it. Beside that we should suppose that the population distribution is a Gaussian, but in most real cases is not a resonable assumption. To solve this problem we can use the **central limit theorem**: If we take a finite number of indipendent finite samples of the population and we compute their mean value (that is a aleatory variable \bar{X}), if the sample size N is sufficiently high then the **sampling means distribution** is gaussian (approximately, and even though the underlying population isn't) which mean $\mu_{\bar{x}}$ is an approximation of the population mean and has standand deviation $\sigma_{\bar{x}} = \frac{\sigma}{\sqrt{N}}$ (called standard error). Greater the sample size, the better the sampling means gaussian distribution's mean approximates the population mean (figure 3.1): at the limit $N \rightarrow +\infty$ then $\bar{x} \rightarrow \mu$ (statistically we can also say $E[\mu_{\bar{x}}] = \mu$) and the standard error $\frac{\sigma}{\sqrt{N}} \rightarrow 0$. [8]

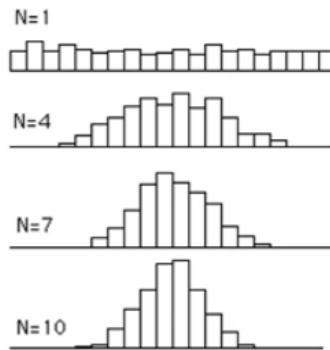


Figure 3.1: Sample mean distribution as the sample size grows

Parameter	Population distribution	Sample	Sampling distribution of \bar{X} 's
Mean	μ	\bar{X}	$\mu_{\bar{x}}$
Standard deviation	σ	s	$\sigma_{\bar{x}} = \frac{\sigma}{\sqrt{n}}$

Figure 3.2: Population, sample and sampling means distribution parameters

Fortunately, it is not necessary to gather too many samples. Infact it is possible to determine the confidence interval from just one sample: if this sample has a sufficient sample size (sperimentally $N \geq 30$), we can approximate the σ (that we don't know) with the sample standard deviation s and compute the confidence interval as follows (at $100(1-\alpha)\%$ significance level) [8]:

$$\bar{x} - z_{\alpha/2} \left(\frac{s}{\sqrt{n}} \right) \leq \mu \leq \bar{x} + z_{\alpha/2} \left(\frac{s}{\sqrt{n}} \right)$$

$$P \left\{ \bar{X} - z_{\alpha/2} \frac{\sigma}{\sqrt{n}} \leq \mu \leq \bar{X} + z_{\alpha/2} \frac{\sigma}{\sqrt{n}} \right\} = 1 - \alpha$$

Figure 3.3: 100(1- α)% Significance level CI

Where \bar{X} is the sampling mean aleatory variable, \bar{x} is the observation of \bar{X} we got from the sample we took, and $z_{\alpha/2}$ is the $\frac{\alpha}{2}$ quantile of the **Standard normal distribution** (that we can use thanks to the Central limit theorem) also known as 100(1- α)% significance level Z-score [8].

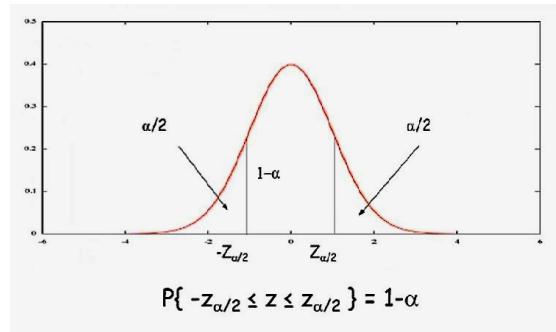


Figure 3.4: Z distribution

3.1.1 Computing WCETs

I've sampled more or less 900 execution times ² for each voter thread (each one computed independently). Using **JMP** Pro (*Predictive Analytics Software* for scientists and engineers ³) i've computed the mean and the standard deviation of the distributions (as shown in figure ...), in order to use it to evaluate the corresponding confidence interval (at 99% significance level) for each thread's average execution time. Note that from each execution times distribution have been eliminated some **outliers**, that can be for sure addressed to the fact that RTAI is executing on a Virtual Machine.

Statistiche di riepilogo	
Media	1883602,6 = 1.883 ms
Dev std	665883,65 = 0.665 ms
Errore std della media	21457,801 = 0.021 ms
Media superiore al 95%	1925712
Media inferiore al 95%	1841493,1

Figure 3.5: Execution times distribution voter thread sensor 1

Statistiche di riepilogo	
Media	30669629 = 30.669 ms
Dev std	437971,51 = 0.437 ms
Errore std della media	14831,587 = 0.148 ms
Media superiore al 95%	30698739
Media inferiore al 95%	30640519

Figure 3.6: Execution times distribution voter thread sensor 2

So for a 100(1- α)% = 99% significance level, we need $\alpha = 0.01$ so $\frac{\alpha}{2} = 0.005$. the corresponding Z-score is $z=2.576$ (this value can be easily found in **z-scores tables** just starting from the desired significance

²Note that the execution time is calculated while only one of the thread is executing and is calculated using slack time because, along with the RTAI modules, on the CPU 1 the thread is the only hard real time thread executing, thus there is no preemption from higher priority tasks.

³JMP is a suite of computer programs for statistical analysis developed by "JMP Statistical discovery", a subsidiary of SAS Institute. It was launched in 1989 to take advantage of the graphical user interface introduced by the Macintosh operating systems. It has since been significantly rewritten and made available also for the Windows operating system.

level, thus from the desired α). So, assuming the sample size as $N=900$, these are the CI evaluated:

$$1.82935589 \text{ ms} \leq \mu_1 \leq 1.937845 \text{ ms}$$

$$30.633951 \text{ ms} \leq \mu_2 \leq 30.707236 \text{ ms}$$

So the WCET of the voter thread for sensor 1 is $WCET_1 = 1.94 \text{ ms}$ with 99% probability, and of the voter thread for sensor 2 is $WCET_2 = 30.7 \text{ ms}$ with 99% probability.

3.1.2 Computing Task-set scheduling feasibility

Like Liu & Lyaland in 1973 demonstrated, the **Least Upper Bound** to the CPU usage for a periodic task-set of n tasks scheduled using RM is equal to (equation n° 4.9 in “*Hard Real-Time Computing Systems: Predictable scheduling algorithms and applications, 3rd edition*” [1]):

$$U_{lub} = n (2^{\frac{1}{n}} - 1) \cong 0.8284$$

In my case the voter thread for sensors of type 1 has period $T_1 = 20 \text{ ms}$ and the voter thread for sensors of type 2 has period $T_2 = 60 \text{ ms}$. Now we can compute the CPU 1 utilization factor, and use the least upper bound theorem for RM to evaluate the task-set scheduling feasibility ⁴:

$$U_{CPU_1} = \sum_i^n U_i = \sum_i^2 \frac{WCET_i}{T_i} = \frac{1.94 \text{ ms}}{20 \text{ ms}} + \frac{30.7 \text{ ms}}{60 \text{ ms}} = 0.6086 \leq 0.8284$$

The CPU 1 utilization factor is way underneath the RM Least Upper Bound thus the task-set is theoretically feasible. Note that, even though the second thread has a computed WCET of 30.7 ms, that is higher than the period of the first thread (20 ms), the first thread is not going in overrun and the task-set is still feasible thanks to the fact that RM is a preemptive scheduling policy (fig 3.7, note that the threads are not submitted to the scheduler at the exact same time, there is a little delay between them, and in the second example is taken it in account).

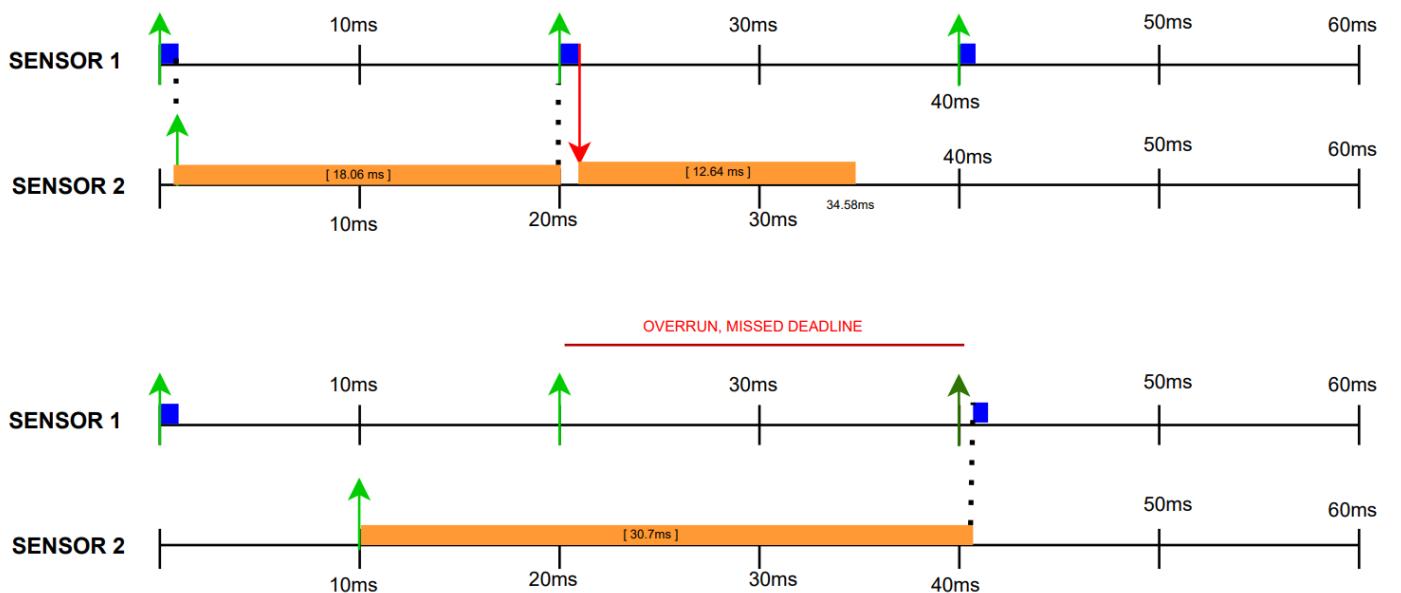


Figure 3.7: Timeline voter execution examples

⁴Note that the voter thread 1 and the voter thread 2 don't have a mutual exclusion access to a same resource, thus the schedulability analysis is way easier because we don't have to consider the eventual delay of blocking on a shared critical session

Chapter 4

Project execution guide

4.1 VM installation

The **OVF archive**¹ (OVA archive) is given along with this pdf document. First of all you should install **Oracle VM VirtualBox** from the official website [9], then use the import functionality (*file*→*import*) to import the disk image along with all the VM configuration informations, in other words to create a full copy of the VM i've used, but in your hypervisor (**the password is "rtai"**).

4.2 Compiling and executing

The project directory is structured as follows:

- **sensor directory:** Along with the sensor subsystem source code it contains "**run**", a script to load the necessary RTAI modules (using "**ldmod**" script²), to execute the sensor susbsystem executable (compiled using the **makefile**) and to remove the RTAI modules previously loaded (using "**remod**" script).
- **voter directory:** Along with the voter subsystem source code it contains the makefiles necessary to compile the voter kernel module, and the header file "**functions.h**" that contains the implementation of the voting methods and of other utility functions such as "**busy_wait_ns**" that implements a parametric busy wait.
- **parameters.h:** this header file contains all the modifiable parameters of the software system (it's shared between the sensor and voter subsystem)
- **README:** a readme file with all the instructions in order to correctly compile and run the system.

4.3 Expected behaviour

Once started the sensor subsystem in one terminal (let's call it terminal_1) and the voter subsystem in another terminal (let's call it terminal_2), we should see in terminal_1 the correct voted measurement for each sensor type, printed periodically (this frequency is a parameter that can be modified in the general header file). See figure 4.1.

¹Open Virtualization Format is an open standard for packaging and distributing virtual appliances or, more generally, software to be run in virtual machines

²Note that the ldmod script sometimes can fail to load the modules. In this case, the only way to fix it is just restart the VM until the problem it's solved.

In terminal_2, after removing the voter module from the RTAI kernel, can be used the command **"dmesg"** to print in the console the cyclic RTAI log content, as shown in figure 4.2.

```

Terminal - root@rtai-VirtualBox:/home/rtai/Scrivania/SC/sensor
File Modifica Visualizza Terminalle Schede Auto
Processed sensor_1: 51 [type1]
Processed sensor_2: 11 [type2]
Processed sensor_1: 53 [type1]
Processed sensor_2: 10 [type2]
Processed sensor_1: 50 [type1]
Processed sensor_2: 11 [type2]
Processed sensor_1: 52 [type1]
Processed sensor_2: 10 [type2]
Processed sensor_1: 51 [type1]
Processed sensor_2: 11 [type2]
Processed sensor_1: 52 [type1]
Processed sensor_2: 11 [type2]
Processed sensor_1: 50 [type1]
Processed sensor_2: 12 [type2]
Processed sensor_1: 50 [type1]
Processed sensor_2: 11 [type2]
Processed sensor_1: 50 [type1]
Processed sensor_2: 10 [type2]
Processed sensor_1: 50 [type1]
Processed sensor_2: 11 [type2]
Processed sensor_1: 53 [type1]
Processed sensor_2: 12 [type2]
Processed sensor_1: 53 [type1]
Processed sensor_2: 12 [type2]
Processed sensor_1: 54 [type1]
Processed sensor_2: 10 [type2]
Processed sensor_1: 50 [type1]
Processed sensor_2: 11 [type2]
Processed sensor_1: 53 [type1]
Processed sensor_2: 11 [type2]
Processed sensor_1: 53 [type1]
Processed sensor_2: 10 [type2]
Processed sensor_1: 52 [type1]
Processed sensor_2: 12 [type2]
Processed sensor_1: 50 [type1]
Processed sensor_2: 11 [type2]
Processed sensor_1: 54 [type1]
Processed sensor_2: 10 [type2]
Processed sensor_1: 50 [type1]
Processed sensor_2: 11 [type2]
Processed sensor_1: 53 [type1]
Processed sensor_2: 11 [type2]
Processed sensor_1: 53 [type1]
Processed sensor_2: 10 [type2]
Processed sensor_1: 52 [type1]
Processed sensor_2: 12 [type2]
Processed sensor_1: 52 [type1]
Processed sensor_2: 11 [type2]
Processed sensor_1: 50 [type1]
Processed sensor_2: 11 [type2]

```

Figure 4.1: Terminal_1

```

Terminal - root@rtai-VirtualBox:/home/rtai/Scrivania/SC/voter
File Modifica Visualizza Terminalle Schede Auto
323.316424| sensors : 49
323.316424| processed sensor_1: 50
323.316424| SLACK SENSOR 2 (voter task): 207772176
323.316424| sensors : 10
323.316424| sensors : 10
323.316424| sensors : 10
323.316424| sensors : 10
323.316424| processed sensor_2: 10
323.316424| SLACK SENSOR 1 (voter task): 188137575
323.316424| sensors : 54
323.316424| processed sensor_1: 54
323.356495| SLACK SENSOR 1 (voter task): 7793436
323.356497| sensors : 54
323.356497| sensors : 54
323.356497| sensors : 54
323.356498| sensors : 54
323.356498| sensors : 54
323.356498| sensors : 54
323.356498| processed sensor_1: 54
323.377652| SLACK SENSOR 1 (voter task): 18875769
323.377652| sensors : 53
323.377652| processed sensor_1: 53
323.377652| SLACK SENSOR 2 (voter task): 20884031
323.377652| SLACK SENSOR 1 (voter task): 18844599
323.377652| sensors : 50
323.377652| processed sensor_1: 50
323.377652| sensors : 10
323.377652| sensors : 10
323.377652| sensors : 10
323.377652| sensors : 10
323.377652| processed sensor_2: 10
323.410184| OVERRUN STATISTICS:
323.410184| sensor 1 overrun: 0
323.410184| sensor 2 overrun: 0
323.410185| CLEANUP MODULE COMPLETED

```

Figure 4.2: Terminal_2

Bibliography

- [1] Giorgio C. Buttazzo. “*hard real-time computing systems: predictable scheduling algorithms and applications, 3rd edition*”. Springer, 2011.
- [2] Dipartimento di Ingegneria Aerospaziale Politecnico di Milano. *rtai 3.4 user manual rev 0.3*. https://www.rtai.org/userfiles/documentation/documents/RTAI_User_Manual_34_03.pdf.
- [3] Dipartimento di Ingegneria Aerospaziale Politecnico di Milano. ”*rtai beginner’s guide*”. <https://www.rtai.org/Beginner's-guide.html>.
- [4] Dipartimento di Ingegneria Aerospaziale Politecnico di Milano. *rtai programming guide v1.0*. http://ppedreiras.av.it.pt/resources/str1213/praticas/rtai_prog_guide.pdf.
- [5] Dipartimento di Ingegneria Aerospaziale Politecnico di Milano. *rtai - real time application interface official website*. <https://www.rtai.org/>.
- [6] Sani Theo (electronicsforu.com). ”*sensors that add strength to aviation and aerospace*”. <https://www.electronicsforu.com/market-verticals/sensors-strength-aviation-aerospace>.
- [7] Javier Diaz Alonso & José Luis Gutiérrez. ”*safety-critical multi-core embedded systems*”. <http://www.ugr.es/~jlgutierrez>.
- [8] Raj Jain. ”*art of computer systems performance analysis techniques for experimental design measurements simulation and modeling*”. Wiley Computer Publishing, John Wiley & Sons, Inc., —.
- [9] Oracle. *oracle vm virtualbox offical website*. <https://www.virtualbox.org/>.
- [10] Jim Anderson & Don Smith. ”*multicore in avionics: enabling heterogeneous accelerators and dynamic workloads*”. U.S. Office of Naval Research, 2021.
- [11] Wikipedia. *rate monotonic scheduling*. https://en.wikipedia.org/wiki/Rate-monotonic_scheduling.