EXTENDING RT-MINIX WITH FAULT TOLERANCE CAPABILITIES

Pablo J. Rogina Gabriel Wainer
{pr6a, gabrielw}@dc.uba.ar

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Pabellón I - Ciudad Universitaria
Buenos Aires (1428) – ARGENTINA

Abstract - The MINIX operating system was extended with real-time services, ranging from A/D drivers to new scheduling algorithms and statistics collection. A testbed was constructed to tests several sensor replication techniques in order to implement and verify several robust sensing algorithms. As a result, new services enhancing fault tolerance for replicated sensors were also provided within the kernel. The resulting OS offers new features such as real-time task management (for both periodic or aperiodic tasks), clock resolution handling, and sensor replication manipulation.

*Index Terms*¾Fault tolerance, Operating Systems, Real-time Systems, Sensing Algorithms, Sensor Replication.

1. INTRODUCTION

Computing systems are already among almost any human activities. In particular, real-time systems (those where the correctness depends not only on the results obtained, but also on the time at which these results are produced) are present in more and more complex tasks every day, where an error can lead to catastrophic situations (even with danger to human life). Therefore, fault tolerance capabilities for this kind of systems are critical to their success during their lifetime cycle. Although fault tolerance strategies are being developed since a long time ago, they were oriented mainly to distributed systems.

This kind of systems span from microcontrollers in automobile engines to very complex applications, such as aircraft flight control or process control in manufacturing plants. Nonetheless, most real-time systems consist of a control system and a controlled system. Information about the environment is provided via sensors, and the system can in turn modify the state of the environment through actuators. Let's take for example a simple manufacturing process: a water tank must have its temperature and pH within a certain range; this is a basic control process (see Fig. 1). The environment is the controlled system, and a computer must keep the temperature and balance the pH. It is necessary that the control system monitors the environment, using sensors (a thermometer and a pH-meter in this case). The control system changes the environment by means of another type of components: actuators (for the example, a heater and an acid injector).

A control process can follow these steps in a repetitive manner, with time constraints applied:

Sensing: real world status must be known (by measuring temperature and pH value)

Controlling: real world values must be checked. Temperature and pH should be within certain limits (lower and upper).

Acting: real world status may need to be changed. Turning the heater on to raise the temperature until the required value is reached.

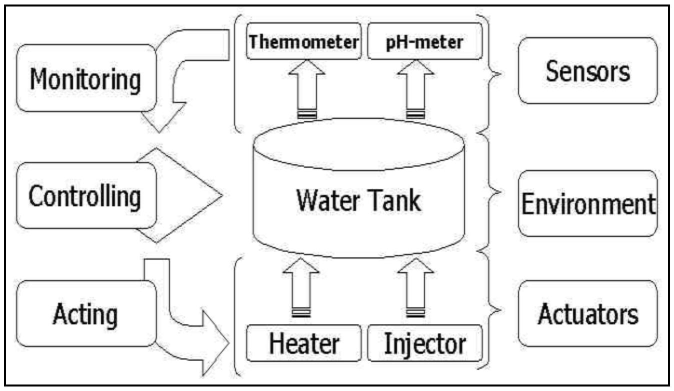

Fig. 1 - Scheme for a basic control process

When real-time systems are built using the services of a programming environment, the timing constraints of the system are usually attached to processes (or *tasks*). The tasks have timing constraints, called *deadlines*, that cannot be missed. Failure in meeting the tasks' deadlines can lead to catastrophic consequences. In the previous example, letting the tank's content to become acid would be a great economic lost.

The goal on designing and building a fault tolerant system is to guarantee that the system will continue working as a whole, even in the presence of faults. Sensors and actuators (hardware) and tasks (software) are potential sources of failures within a real-time system. The service delivered by a system is the system behavior as it is perceived by another special system(s) interacting with the considered system: its user(s) [1].

Using this definition, it can be said that a system *faults* when it fails to deliver the service(s) it is intended to. Depending on the system's complexity and relevance, this failure can be tolerated (statistical erroneous data from a census, that can be recalculated again later) or can lead to a catastrophic accident (an air traffic control system). The broader use of computers in critical missions forced the need to improve the capacity of avoiding and tolerate faults. A failure is an error, due perhaps to a design problem, manufacturing, programming, a human error or environmental conditions. A component failure generally does not lead straightly to the failure of the whole system, but it may be the beginning of a number of failures ending in the system's fault.

Failures can occur due to errors in the hardware (a short-circuit) or errors in the software (using '!=' instead of '=' in a C program). The first case, known as hardware fault tolerance, is well-understood, to the point of being an Engineering discipline. Several reasons can be cited:

- The physics of hardware components, such as silicon, are well understood. The complexity of large hardware designs is several orders of magnitude less than large software systems.
- Given the costs associated with mass production, hardware engineers produce carefully thought out specification along with functional tests that can be applied in order to test units coming off the assembly line.
- Electronic components are more reliable year after year. Values of MTBF (Mean Time Between Faults) have raised continuously in the last decades. No one can imagine a hard disk with faulty sectors these days. Fault tolerant systems are expected to go on (survive) even with component faults, not to rely on the low probability of them to fail.

Timing faults can be classified in:

Transient	they happen once and then disappear. If the task or action is repeated, the fault does not occur again.
Intermittent	they appear, disappear, and are present again. This condition makes them hard to diagnose.
Permanent	faults that are present until the failed component is replaced or repaired.

This work is devoted to present the efforts in building a programming environment for real-time systmes. The work is based on a modification of the Minix operating system, so as the results can be used with educational purposes. Sensor replication schemes were included in the kernel, providing fault tolerance when sensing values from the real world.

The work is organized as follows: Section 2 describes the extensions done to the RT-MINIX operating system. Section 3 is devoted to fault tolerance capabilities related with sensing algorithms and sensor replication; while sensing algorithms are presented in Section 4. Both static and dynamic tests are discussed in Sections 5 and 6 respectively. Finally, conclusions and future work proposals are listed in Section 7.

2. REAL-TIME EXTENSIONS TO MINIX

As many other computer applications, real-time systems are usually built by using the services offered by an operating system. In this case, those services should be slightly different than the case for traditional applications. It should provide basic support for predictability, satisfaction of real-time constraints, fault tolerance and integration between time-constrained resources and scheduling.

Existing real-time operating systems (RTOS) can be divided in two categories:

1. Systems implemented using somewhat stripped down and optimized (or specialized) versions of conventional timesharing OS
2. Systems starting from scratch, focusing on predictability as a key design feature.

The MINIX 1.5 operating system [2] was taken as a base, and it was extended with several real-time services [3]. The most important include task management capabilities (both for periodic an aperiodic tasks) and real-time scheduling algorithms (Rate Monotonic and Earliest Deadline First). These strategies were later combined with other traditional strategies, such as Least Laxity First, Least Slack First and Deadline Monotonic. At present new flexible schedulers are being included.

To allow these changes several data structures in the operating system were modified (to consider tasks period, execution time and criticality). A new multi-queue scheme was defined, so as to accommodate real-time tasks along with interactive and CPU-bound tasks. The original task scheduler of MINIX used three queues, in order to handle task, server and user processes in that order of priority. Each queue was scheduled using the Round Robin algorithm.

A new set of signals was added to indicate special situations, such as missed deadlines, overload or uncertainty of the schedulability of the task set. All these services were made available to the programmer as a complete set of new system calls. A long list of tests demonstrated the feasibility of MINIX as a workbench for real-time development.

Several work was done using the tool, spanning from the testing of new scheduling algorithms to kernel modifications. Recently, the need to integrate the previous work in a new version for the operating system arisen. This was motivated in part for the release of new MINIX versions in the meantime, and because several additional features were identified that would be useful to be added to original environment.. Those extensions [4] were done using MINIX 2.0; include the previous services and add new ones such as analog-digital conversion, queue model modification and new real-time metrics. These services are described in detail in the following paragraphs.

The need to acquire analogic data from the environment motivated this new feature. As stated before, many real-time systems are used to control a real process, such as a production line or a chemical reaction. A device driver was written following the same framework used under Linux [5], with slight changes. The device driver adds a new kernel task that provide the programmer with three basic operations (open, read, close) to access an A/D converter as a character device (for instance, /dev/js0 and /dev/js1, for joystick A and joystick B, respectively).

A second set of changes was related with the task scheduler management. The ready process queuing and handling is arranged in four levels. The basic idea considered in joining the queues was related with the goal that a real-time task should not be interfered by low level interrupts (and its associated servers), working with the hypothesis that server and user queues can be joined, allowing File System (FS) and Memory Manager (MM) processes to be moved from server to user process category. An in-depth analysis was made to check the possibility of deadlock between FS and MM, first revisiting the semantics of them and then trying to measure the impact of the new scheduler (with the joined queues), showing that deadlocks cannot occur with the changed scheduler.

Once the OS was extended with real-time services, the need arose to have several measuring tools. It is needed to test the evolution of the executing tasks according with the different scheduling strategies. The impact of the different workloads should be also considered. To do so, the kernel is in charge to keep a new data structure accessible to the user via a system call. Statistics also can be monitored online by means of a function key displaying all that information.

MINIX proved to be a feasible testbed for OS development and real-time extensions that could be easily added to it. This "new" operating system (a MINIX 2.0 base with real-time extensions) has a rich set of features, which makes it a good choice to conduct real-time experiences. The added real-time services covered several areas:

Task creation: tasks can be created either periodic or aperiodic, stating their period, worst execution time and priority

Clock resolution management: the resolution (grain) of the internal clock can be changed to get better accuracy while scheduling tasks.

Scheduling algorithms: both RMS and EDF algorithms are supported, and can be selected on the fly.

Statistics: several variables about the whole operation are accessible to the user to provide data for benchmarking and testing new developments.

Supervisory Control and Data Acquisition: as a user application, it makes full use of real-time services.

3. FAULT TOLERANCE CAPABILITIES

To avoid systems being vulnerable to a single component failure, it is reasonable to use several sensors redundantly; this is, using one of the more broad used fault tolerance technique: replication. Let's think of an automatic tracking system: it could use different kinds of sensors (radar, infrared, microwave) that are not vulnerable to the same kinds of interference. However, redundancy presents a new problem to system designers because the system can receives several readings that are either partially or entirely in error. To improve sensor-system reliability, the practical problem of combining, or *fusing*, the data from many independent sensors into one reliable sensor reading has been widely studied. The principal goal is to provide the application with the ability to make the correct decision in the presence of faulty data.

Much will depend on the system's accuracy (the distance between its results and the desired results) and the system's precision (the size of the value range it returns). As sensors employed in real-time systems are inherently unreliable, distributed sensors makes reliability even compromised.

In [6], a set of robust sensing algorithms are revised and a new hybrid algorithm is presented. The proposed new algorithm is a combination of other two: inexact agreement and optimal region. The new mechanism provides more accuracy and precision. The solution is derived from independent sources: one is based on set theory, the other in geometry, producing two explanations of the same problem.

With the aim to prove those proposed solutions, a model with replicated sensors was implemented, and the platform of choice was RT-MINIX. This OS allows to connect a set of sensors using different input methods. The following sections will be devoted to analyze the basic properties regarding this new capabilities.

The new capabilities of RT-MINIX regarding the joystick driver, allowed to connect a set of "sensors" in the form of potentiometers to the game port. First of all, user applications were written to validate the concepts, and the better ones were coded into the OS kernel.

4. SENSING ALGORITHMS

The algorithms selected to be implemented under RT-MINIX were taken from [6], and are described below:

```
Algorithm:   Approximate-agreement
Input:       A set of sensors, each with a value.
Output:      A set of sensors, each with a new value converging toward
             a common value.

Step 1:      each sensor broadcasts its value.
Step 2:      each sensor receives the values from the other sensors and
             sots the values into vector $v$.
Step 3:      the lowest $\tau$ values and the highest $\tau$ values are discarded
             from $v$ at each sensor.
Step 4:      each sensor forms new vector $v'$ by taking the remaining
             values $v[i*\tau]$ where $i$=0,1,... (the smallest remaining value
             and every remaining $\tau$'th value in order).
Step 5:      the new value is the mean of the values in $v'$.
```

```
Algorithm:   Fast Convergence
Input:       a set of sensors, each with a value.
Output:      A set of sensors, each with a new value converging toward
             a common value.

Step 1:      each sensor receives the values from all other sensors and
             forms set $V$.
Step 2:      acceptable values[1] are put into a set $A$.
Step 3:      e($A$) is computed.
Step 4:      any unacceptable values are replaced in $V$ by e($A$) [2].
Step 5:      the new sensor value is the average of the values in $V$.
```

```
Algorithm:   Optimal Region
Input:       a set of sensor readings $S$.
Output:      a region describing the region that must be correct.
```

[1] A value is acceptable if it is within distance δ of N-τ other values.

[2] e(A) can be any of a number of functions on the values of A. The authors suggested average, median, or midpoint as possible choices of e(A) that may be appropriate for different applications.

Step 1: initialize a list of regions, called *C*, to NULL.
Step 2: sort all points in *S* into ascending order.
Step 3: a reading is considered active if its lower bound has been traversed and its upper bound has yet to be traversed. Work through the list in order, keeping track of active readings. Whenever a region is reached where *N-t* or more readings are active, add the region to *C*.
Step 4: All the points have been processed. List *C* now contains all intersections of (*N-t*) or more readings. Sort the intersections in *C*.
Step 5: output the region defined by the lowest lower bound and the largest upper bound in *C*.

Algorithm: *Brooks-Iyengar Hybrid*
Input: a set of data *S*.
Output: a real number giving the precise answer and a range giving its explicit accuracy bounds.

Step 1: each sensor receives the values from all other sensors and forms set *V*.
Step 2: perform the optimal region algorithm on *V* and return a set *A* consisting of the ranges where at least *N-t* sensors intersect.
Step 3: output the range defined by the lowest lower bound and the largest upper bound in *A*. These are the accuracy bounds of the answer.
Step 4: sum the midpoints of each range in *A* multiplied by the number of sensors whose readings intersect in that range, and divide by the number of factors. This is the answer.

A sensor is called a processing element (PE). The number of PEs is N and τ is the number of malfunctioning PEs. These algorithms are intended to return a valid value from a set of readings from N PEs given τ of them are known (or supposed) to be wrong; not to establish how many sensors are faulty.

5. STATIC TESTS

To prove that the algorithms have been implemented properly, a set of tests had to be conducted. At a first step, data was used "statically", this is, hard-coded in the test programs. The set of values used in the first test were the same presented in [6] and shown in Table 1. It simulates a set of 5 sensors, one of them working in a faulty manner, thus providing a different value each time a reading was made. This set of sensors can be thought as belonging to a robotic arm, providing information about the arm's elbow position, for example. The measured angle is expressed as a value along a tolerance (both plus and minus). Those ranges imply the concept of *abstract sensor*: "a set of values that contains the physical variable of interest" [7].

Case	**S 1**	**S 2**	**S 3**	**S 4**	**S 5**
1	4,7 ± 2,0	1,6 ± 1,6	3,0 ± 1,5	1,8 ± 1,0	3,0 ± 1,6
2	4,7 ± 2,0	1,6 ± 1,6	3,0 ± 1,5	1,8 ± 1,0	1,0 ± 1,6
3	4,7 ± 2,0	1,6 ± 1,6	3,0 ± 1,5	1,8 ± 1,0	2,5 ± 1,6
4	4,7 ± 2,0	1,6 ± 1,6	3,0 ± 1,5	1,8 ± 1,0	0,9 ± 1,6

Table 1 - Sensors and its broadcasted values [6]

Each one of the algorithms shown above were applied to all the four cases in Table 1. At any time, the number of sensors is 5, and the number of sensors with intermittent failures is 1. These conditions preserve the effectiveness of the algorithms (because 1<5/2). Results achieved by our own version of the algorithms running under RT-MINIX were the same stated in [6], thus validating our implementation.

The algorithms were also tested using another set of values, this time taken from [8]. Fig. 2 shows both the set of values and the results to be obtained.

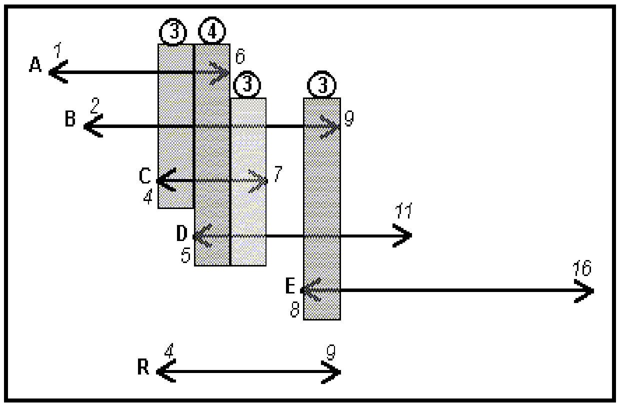

Fig. 2 - Values and regions [8]

In this example, sensors are represented by arrows (labeled with letters from A to E), with values once again expressed as ranges (indicated by numbers on both arrows' ends). The shaded rectangles are regions that Optimal Region and Brooks-Iyengar algorithms have to identify, where the circled numbers above the regions represent the number of intersections in that region. Finally, arrow R is the interval where the answer should be found. All algorithms were applied to this set of values, and their output is shown in Fig. 3.

```
Testing robust sensing algorithms with static data

Approximate Agreement Alg.:  6.33
Optimal Region Alg.       :  [4.0..9.0]
Brooks-Iyengar Hybrid Alg.:  [4.0..9.0]  6.192
Fast Convergence Alg.     :  6.90
```

Fig. 3 - Output from second static test

6. DYNAMIC TESTS

After the algorithms have been successfully proven with static data, an idea took form in the manner to prove them once again, this time with dynamic data, i.e. variable from test to test.

To provide the algorithms with such sets of values, a device was built: four linear 100MΩ potentiometers were connected to each one of the four resistive inputs on the game port of a PC. This testbed would use one of the recent real-time services available in RT-MINIX (Analogic/Digital conversion capabilities through the joystick driver). The potentiometers can be thought this time as sensors for a valve in a pipeline, providing information about the valve position, where the minimum value referring the valve as totally closed, while the maximum value representing the valve as totally open. The wiring diagram for the testbed is shown in Fig. 4.

An auxiliary program was written to read the four inputs simultaneously, showing the values on screen. This application is used to adjust the "sensors" to the desired value, allowing to simulate a faulty one; positioning it out of range from the remaining ones (for this test, N=4 and τ=1).

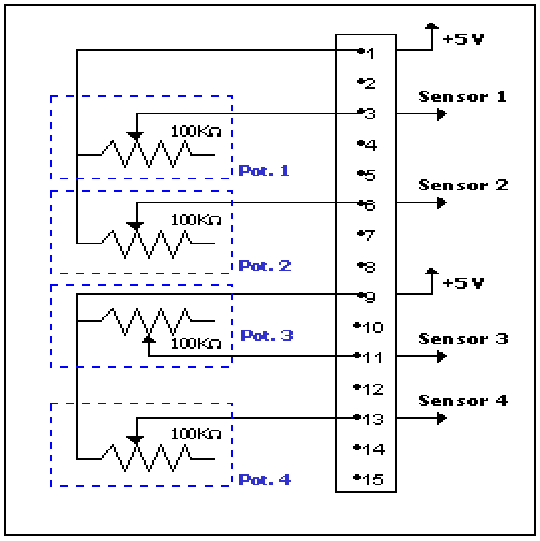

Fig. 4 - Testbed's wiring diagram

After the model is adjusted to a particular situation, the main test program is run. At first, a set of readings are taken from the model. A *sensor reading* is defined as a value along with a lower bound and an upper bound. Thus, to make a sensor reading, three consecutive port readings are made, repeating this process for each of the four sensors. Each of the available algorithms are then applied to this set of sensor readings, displaying the results on screen (see Fig. 5).

```
Testing robust sensing algorithms with dynamic data

Sensor  L. Bound        Value           U. Bound
  0     578.0           638.0           677.0
  1     614.0           626.0           688.0
  2     312.0           314.0           316.0
  3     604.0           649.0           681.0

Approximate Agreement Alg.: 632.00
Optimal Region Alg.       : [614.00..677.00]
Brooks-Iyengar Hybrid Alg.: [614.00..677.00] 645.50
Fast Convergence Alg.     : 556.75
```

Fig. 5 - Output from dynamic test

After the implementation steps and tests were finished, some comparisons could be drawn:

- *Development*: none of the algorithms imposed difficulties in their implementation.
- *Response time*: no evident differences in response time from all the algorithms were found.
- *Results*: *Approximate Agreement* (AA) and *Fast Convergence* (FC) return a value, while *Optimal Region* returns a range, and *Brooks-Iyengar Hybrid* returns a range plus a value. *Optimal Region* (OR) and *Brooks-Iyengar Hybrid* (BIH) give answers within a narrower range than input data. As several dynamic tests were performed, with the model adjusted to different situations, it was found that the answer from *Approximate Agreement* always fell inside the range returned from OR and BIH, while the broader the range, more the difference between this value and the answer from BIH.

7. CONCLUSION

Fault tolerance, as a key discipline with growing use inside real-time systems, provides several techniques and schemes that can and must be used in different areas of such systems: from specification languages and temporal logic in the definition steps; the scheduling perspective and replication of sensors and actuators in the implementation steps.

This work described how the real-time extensions to the MINIX operating system, transforming it into RT-MINIX, have been complemented with fault tolerant sensing algorithms to allow the development of applications taking benefits of that kind of services provided from the operating system kernel. With these extensions, RT-MINIX can be used as a platform for real-time processing or as a starting point for adding more real-time services. Robust sensing algorithms were implemented and tested under RT-MINIX, and are now available as a service to applications having to deal with sensor replication.

Future work may include extending the sensing algorithms to deal with multidimensional sensors, (replacing each interval corresponding to a physical value by a vector of intervals). Fault Tolerant schedulers must be studied and integrated in a next version of RT-MINIX, providing the programmer with a specialized and improved fault-tolreant environment.

8. ACKNOWLEDGEMENTS

This work was partially supported by the UBA-SECYT research project TX-004, "Concurrency in Distributed Systems".

All the related source code can be obtained at http://www.dc.uba.ar/people/proyinv/cso/rt-minix together with downloading and installation instructions.

9. REFERENCES

[1] J. Laprie, "Dependable Computing and Fault Tolerance: Concepts and Terminology", 15th Annual Int. Symposium on Fault-Tolerant Computing, pp 2-11, June 1985.

[2] A. Tannenbaum, "A Unix clone with source code for operating systems courses", *ACM Operating Systems Review*, 21:1, January 1987.

[3] G. Wainer, "Implementing Real-Time Scheduling in a Time-Sharing Operating System", *ACM Operating Systems Review*, July 1995.

[4] P. Rogina and G. Wainer, "New Real-Time Extensions to the MINIX operating system", Proc. of 5th Int. Conference on Information Systems Analysis and Synthesis (ISAS'99), August 1999.

[5] V. Paulik, "Joystick device driver for Linux", source code and installation details available online at ftp://atrey.karlin.mff.cuni.cz/pub/linux/joystick/joystick-0.8.0.tar.gz

[6] R. Brooks and S. Iyengar, “Robust Distributed Computing and Sensing Algorithm”, *IEEE Computer*, pp 53-60, June 1996.

[7] K. Marzullo, “Tolerating failures of continuous-valued sensors”, *ACM Transactions on Computer Systems*, 8(4):284-304, November 1990.

[8] D. Jayasimha, “Fault Tolerance in a Multisensor Environment”, Dept. of Computer Science, The Ohio University, May 1994.