Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

INFORME TÉCNICO

Extending MINIX with Real-Time Services and Fault Tolerance Capabilities

Pablo J. Rogina
Dr. Gabriel Wainer

Report n.: 2000-001

Pabellón 1 - Planta Baja - Ciudad Universitaria
(1428) Buenos Aires
Argentina

http://www.dc.uba.ar

Title: Extending MINIX with Real-Time Services and Fault Tolerance Capabilities

Authors: Pablo J. Rogina, Dr. Gabriel Wainer

E-mail: pr6a@dc.uba.ar, gabrielw@dc.uba.ar

Report n.: 2000-001

Key-words: Fault Tolerance, Operating Systems, Real-time Systems, Sensing Algorithms, Sensor Replication

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.

To obtain a copy of this report please fill in your name and address and return this page to:

Infoteca
Departamento de Computación - FCEN
Pabellón 1 - Planta Baja - Ciudad Universitaria
(1428) Buenos Aires - Argentina

TEL/FAX: (54)(11)4783-0729
e-mail: infoteca@dc.uba.ar

You can also get a copy by anonymous ftp to: **zorzal.dc.uba.ar/pub/tr**

or visiting our web: **http://www.dc.uba.ar/people/proyinv/tr.html**

Name:.....

Address:.....

.....

Extending MINIX with Real-Time Services and 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—Fault 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.

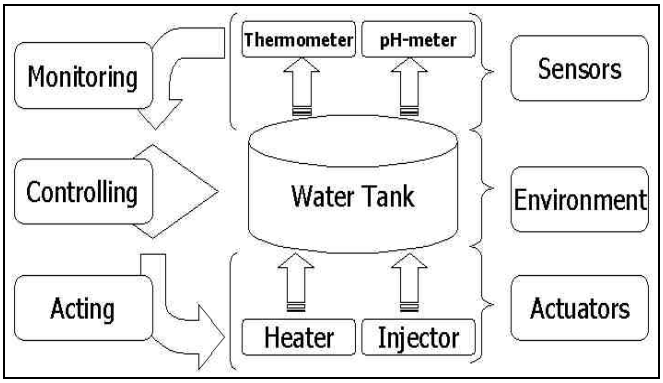

Fig. 1 - Scheme for a basic control process

As many other computer applications, real-time systems are usually built by using the services offered by an operating system. In this case, the services provided 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. 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 systems. 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 present document 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. Sections 6 and 7 explain current applications and work in progress using RT-MINIX. Finally, conclusions and future work proposals are listed in Section 8.

2. REAL-TIME EXTENSIONS TO MINIX

MINIX [2] (name that stands for Mini-UNIX), is a complete, timesharing, multitasking operating system. Inspired by UNIX, it was written from scratch by A. Tannenbaum. Though it is copyrighted, the source has been made widely available to universities for study and research in computer science courses.

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

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

Research projects falling in the first category include KURT [3], RT Mach [4] and RT-Linux [5]. Operating systems like Spring [6], Maruti [7] and YARTOS [8] were developed using the second approach. Though several commercially available systems, including LynxOS [9] and QNX [10], offer real-time performance and services to applications, they are too costly and proprietary to be used by research or academic institutions.

Task *scheduling* in multitasking systems has been extensively studied in the operating system literature. Nevertheless, the traditional scheduling techniques used in general purpose systems (e.g. FIFO, Shortest Job First, Round Robin, etc.) are not adequate to be used in time constrained systems. These scheduling policies attempt to reduce certain performance metrics (i.e., the average response time), and do not deal with the timing constraints of the processes to be scheduled. On the other hand, scheduling policies for real-time systems need to guarantee that tasks will meet their deadlines in all circumstances. Such a set of tasks is called *schedulable*, with each task having a *predictable* behavior. Scheduling algorithms can be divided in two major models:

preemptive, and non-preemptive. The first one assume that any task can be interrupted during its execution, while non-preemptive algorithms do not allow a running task to be interrupted.

Most scheduling algorithms divide the schedulable tasks into two different classes: periodic and aperiodic (sporadic). The periodic tasks must run repeatedly, and within fixed times (known as period). The aperiodic tasks run sporadically, and only once, when they are invoked. Two well-known policies are broadly accepted for Real-Time scheduling: RMS (Rate Monotonic Scheduling) was shown to be optimal for scheduling fixed priority task sets. In dynamic priority systems, using EDF (Earliest Deadline First) policy, full processor utilization can be achieved. Real-time scheduling algorithms are a field of continuing research.

Taking this base into account, the present project shows the results obtained building a new version of an extended Real-Time operating system. MINIX 1.5 was taken as a base, and it was extended it with several real-time services. The most important include task management capabilities (both for periodic an aperiodic tasks), real-time scheduling algorithms; new device drivers allowing A/D conversion, and improved fault tolerance features, specially, robust sensing algorithms incorporated inside the kernel.

The work presented in [11] showed the results obtained in a research project devoted to use MINIX to implement real-time scheduling. Several changes was made to source code of the kernel, in order to provide the user with a set of system calls to create and manage tasks, both periodic or aperiodic. The project was devoted to provide programming facilities to develop hard real-time software. Under the changed MINIX OS, the programmer was allowed to define timing constraints for the tasks, letting the OS to execute them in a timely fashion. In this way, productivity, security and development costs can be improved.

Several real-time services were added. First, RM and EDF scheduling were included. 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). 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 multiqueue scheme was defined, so as to accommodate real-time tasks along with interactive and CPU-bound tasks. 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. In despite of this fact, several additional features were identified to be added to original environment.

Recently, the need to integrate the previous work in a new version for the operating system arisen. This happened because new MINIX versions were released in the meantime. Some of those extensions are presented in the following paragraphs.

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 [12] 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.

A. Analogic-Digital Conversion

The first group of changes was related with the need to acquire analogic data from the environment. As stated earlier, many real-time systems are used to control a real process, such as a production line or a chemical reaction. This implies a ‘sense and act’ attitude, i.e., sensing the environment and then changing it if necessary to keep control of the whole process. To sense the real world, a long list of sensors can be used, ranging from thermometers, pressure, infrared, etc.; many of them providing analogic signals.

The game port interface in the PC allows connecting up to four analog and four digital inputs. Providing the OS with the ability to directly read the game port enhances the chance to connect different analog sensors. The possibility to use this feature from within MINIX was tested [13], and a device driver for the game port was written.

When the new solutions were tested, it showed poor performance when doing the readings. The device driver had to be completed rewritten, this time following the same framework used under Linux [14], with slightly changes. Resistive inputs (coordinates XY) and digital inputs (buttons) are aligned together in a byte (8 bits) that can be read at address 201h. Input pins from D-connector relates with that byte as shown in Fig. 2.

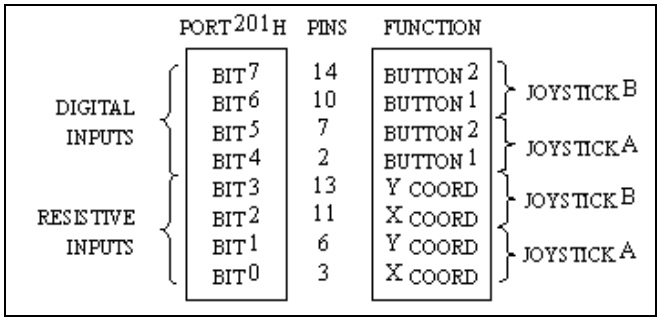

Fig. 2 - Game Port Data bus and pins correlation

The device driver adds a new kernel task that provide the programmer with three basic operations (open, read, close) to access the game port as character devices (/dev/js0 and /dev/js1, for joystick A and joystick B, respectively). To read the axis, the task sends any value to that port (201h) and cycles reading the port, waiting for any of the resistive inputs to become 0. The number of times the cycle is run is proportional to the resistance (and thus position) of the joystick. Some scripts were also modified to make device creation a simple step. At present we are working into the addition of new drivers for different A/D – D/A controllers.

B. Joined Scheduling Queues

A second set of changes was related with the task scheduler management. 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.

Level 3	INIT	User 1	User 2	User n
Level 2	Memory Manager (MM)		File System (FS)	
Level 1	Disk Task	Clock Task	Printer Task	Other Tasks
Level 0	Process Manager			

Fig. 3 - Processes structure in MINIX [2]

The MINIX structure related to processes, message passing architecture and the ready process queuing and handling is shown in Fig. 3. Each of these levels are described as follows:

- *Level 0* is in charge of three fundamental duties: process management; message passing and interrupt management.
- *Level 1* includes I/O processes or *tasks* (known also as device drivers).
- *Level 2* contains only two processes, FS and MM, bringing an extended machine able to manage system calls of certain complexity.
- *Level 3* comprises all the processes below the INIT process, the place for applications (like compilers, shell, editors) and user processes.

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). The work presented in [15] worked on 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.

The expected result of such change is getting better response time from the operating system. The union of the queues avoids interference of the Operating System tasks in the most critical real-time tasks. Several examples of possible scenarios are introduced. Through these case studies and their impacts in processing time, it became clear that the unification was feasible. Reducing the number of queues is also a step towards fault tolerance.

When the availability of shared resources (such as FS or MM) are diminished, a deadlock problem is likely to appear quite often. A deadlock occurs whenever a process is blocked waiting for a second process, while the later is also waiting for the first one.

Under the original scheduler in MINIX 2.0, a process requiring a service from FS or MM had it delivered immediately. This was that way because FS or MM had enough priority to start at any time without being preempted. 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).

The only possible communication between FS y MM (under the original source code) is done during system initialization, and that connection is unidirectional, thus avoiding the circular waiting case. A conclusion from that scheme is that FS and MM work independently, having relation only with processes of task category (the kernel itself or device drivers). Task level processes have higher priority and are not preempted because of that condition, with their execution being considered instantaneous (and atomic) regarding a user process.

The final conclusion is that deadlocks are not probable to occur due to the changed scheduler. User processes cannot communicate each other; FS does not communicate with MM; and the management of the task queue was not altered from the original code. This is a very good feature to achieve fault tolerance.

C. Real-Time Metrics

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.

```
struct rt_globstats {
  int actpertsk;
  int actapetsk;
  int misperdln;
  int misapedln;
  int totperdln;
  int totapedln;
  int gratio;
  clock_t idletime;
};
```

Fig. 4 - Data structure to keep real-time metrics

To do so, the kernel is in charge to keep a data structure that is accessible to the user via a system call. Statistics also can be monitored online by means of a function key displaying all that information on screen. The data structure is shown in Fig. 4 and its items are described as follows:

actpertsk, acrtapetsk: number of active (running) real-time tasks, both periodic and aperiodic.

misperdln, misapedln: number of missed deadlines, both periodic and aperiodic.

totpertsk, totapetsk: number of total scheduled real-time tasks instances, both periodic and aperiodic.

gratio: guarantee ratio, i.e., the relationship between number of instances and deadlines met.

idletime: time (in clock ticks) not used as compute time.

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 [16], 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.

D. Replicated Sensors

Sensor replication is an area of growing interest in real-time processing. It enhances the fault tolerance potential of the whole system by exploiting redundancy. As earlier explained, MINIX has been expanded with sensor reading capabilities, and the existing serial and parallel ports can be connected to data acquisition hardware. The main goal was to include standard fault tolerant strategies, allowing to check the validity of different available sensing algorithms.

The work presented in [17] introduces an important concept in order to tolerate sensor failure: the use of *abstract sensors*. An abstract sensor is a set of values that contains the present value of a physical variable of interest. Each abstract sensor is implemented using a *concrete sensor* (a physical device that reads a physical variable, i.e. a thermometer). The concrete sensor does not need to sense the physical variable of interest. For example, a temperature abstract sensor can be constructed using a manometer to sense pressure and then applying the Boyle's law.

Another important aspect of sensor replication is the ability to enhance the expected accuracy from a set of replicated sensors far beyond the obtainable using only one sensor. This leads to multisensor environments or the use of a distributed network of sensors. Data coming from the physical system may be faulty due to sensor's failure, communication problems or noise. When using sensor replication, a method to combine data from several different sensors is needed. This action is called information integration, and it can be *competitive* or *complementary*.

In the first approach, each sensor theoretically provides identical information (though this is not the case in practice). Complementary information integration is done when partial information is available from each sensor: that information is combined to get the necessary knowledge about the environment.

Another advantage provided by the concept of abstract sensor is the capacity of data abstraction. A strategy of fault tolerance algorithms is to employ different kinds of redundant sensors. Thus, a real application could arrange different

sensors (i.e., infrared, microwaves, and radar) that are not vulnerable to the same type of interference. To specify such a real-time system, only abstract sensors are considered, without concern of the type.

Using the algorithms studied under [16], the idea was to extend RT-MINIX with the possibility to use several sensors from a fault tolerance perspective. First of all, the four algorithms were coded as a user application. The next step was to incorporate the ability to use real data. In this case, the environment was sensed by means of four potentiometers (using the four analogic inputs from the joystick port). The inputs were arranged as a set of concrete sensors (acting as position sensors for a simulated robotic arm).

The algorithms worked as expected, providing a unique value from the replicated sensors and although one of them were faulty (the user had the chance to change data varying the potentiometers as desired).

Finally, the algorithms were combined within the kernel, providing the programmer with a set of functions to work with abstract sensors. It is possible to create (indicating physical devices, such as /dev/js0 and type of algorithms) and then read an abstract sensor, even in the presence of faulty concrete sensors.

4. SENSING ALGORITHMS

The algorithms selected to be implemented under RT-MINIX were taken from [16], 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 τ values and the highest τ values are discarded from *v* at each sensor.
Step 4: each sensor forms new vector *v'* by taking the remaining values *v*[i*τ] where *i*=0,1,... (the smallest remaining value and every remaining τ'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.

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 $N-\tau$ 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 ($N-\tau$) 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 $N-\tau$ 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.

---

[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.

5. TESTING THE ALGORITHMS

E. 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 [16] 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" [17].

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 [16]

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 [16], thus validating our implementation.

The algorithms were also tested using another set of values, this time taken from [18]. Fig. 5 shows both the set of values and the results to be obtained.

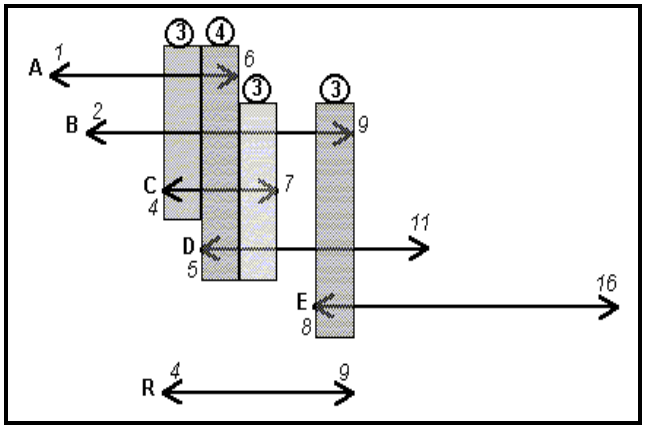

Fig. 5 - Values and regions [18]

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. 6.

```
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. 6 - Output from second static test

F. 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. 7.

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).

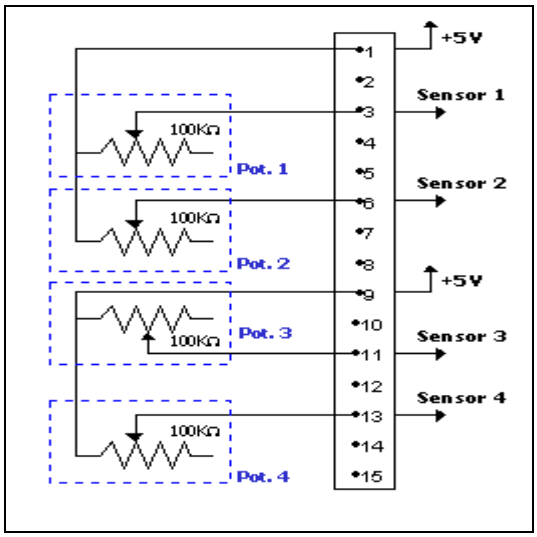

Fig. 7 - Testbed's wiring diagram

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. 8).

```
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. 8 - Output from dynamic test

G. Dynamic Test Summary

As stated earlier, the model provided the ability to set different sensor conditions, making it easy to conduct several runs for the dynamic test.

Run	Sensors					
	1			2		
	lb	v	ub	lb	v	ub
A	14,0	15,0	15,0	683,0	684,0	684,0
B	578,0	638,0	677,0	614,0	626,0	688,0
C	473,0	480,0	507,0	480,0	504,0	538,0
D	492,0	503,0	507,0	433,0	506,0	517,0
E	506,0	516,0	546,0	488,0	517,0	520,0
F	535,0	610,0	703,0	519,0	616,0	660,0
G	675,0	686,0	709,0	658,0	682,0	682,0
H	565,0	638,0	688,0	592,0	658,0	706,0
I	555,0	631,0	702,0	532,0	656,0	759,0
J	651,0	667,0	685,0	625,0	706,0	730,0
K	678,0	679,0	680,0	679,0	680,0	681,0
L	678,0	679,0	682,0	681,0	681,0	681,0

Table 2 - Values used in dynamic test (sensors 1 and 2)

The values used in all the runs are presented in Table 2 and Table 3, showing for each sensor the corresponding reading, in three columns: a lower bound (lb), a value (v) and an upper bound (ub).

The results obtained after applying each algorithm available under RT-MINIX to those readings are contained in Table 4, and are expressed depending on the algorithm, as a value (v); a range with a lower bound (lb) and upper bound (ub) or a range and a value. The algorithms are identified by their initials: AA, Approximate Agreement; OR, Optimal Region; BIH, Brooks-Iyengar Hybrid and FC, Fast Convergence.

Run	Sensors					
	3			4		
	lb	v	ub	lb	v	ub
A	683,0	684,0	684,0	683,0	683,0	684,0
B	313,0	314,0	315,0	604,0	649,0	681,0
C	307,0	308,0	308,0	478,0	492,0	550,0
D	308,0	308,0	308,0	482,0	529,0	531,0
E	307,0	307,0	308,0	480,0	510,0	513,0
F	148,0	218,0	308,0	525,0	609,0	680,0
G	148,0	148,0	148,0	674,0	688,0	689,0
H	148,0	179,0	202,0	609,0	647,0	705,0
I	167,0	193,0	201,0	553,0	610,0	657,0
J	191,0	192,0	192,0	647,0	668,0	686,0
K	193,0	193,0	194,0	672,0	679,0	680,0
L	194,0	194,0	194,0	677,0	679,0	681,0

Table 3 - Values used in dynamic test (sensors 3 and 4)

Run	Algorithms						
	AA	OR		BIH			FC
	v	lb	ub	lb	ub	v	v
A	683,5	683,0	684,0	683,0	684,0	683,2	516,5
B	632,0	614,0	677,0	614,0	677,0	645,5	556,7
C	486,0	480,0	507,0	480,0	507,0	493,5	446,0
D	504,5	492,0	507,0	492,0	507,0	499,5	461,5
E	513,0	506,0	513,0	506,0	513,0	509,5	462,5
F	609,5	535,0	660,0	535,0	660,0	597,5	475,0
G	682,0	675,0	682,0	675,0	682,0	678,5	551,0
H	642,5	609,0	688,0	609,0	688,0	648,5	530,5
I	620,5	555,0	657,0	555,0	657,0	606,0	522,5
J	667,5	651,0	685,0	651,0	685,0	668,0	558,2
K	679,0	679,0	680,0	679,0	680,0	679,5	557,7
L	679,0	678,0	681,0	678,0	681,0	679,5	558,2

Table 4 - Results from all runs at the dynamic test

H. Algorithm Comparison

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 AA always fell inside the range returned from OR and BIH. With these results in view, any of the algorithms could be used. However, there was found that the broader the result range from BIH, more the difference between the result value of that algorithm and the answer from AA.

Test	*Range Amplitude BIH (2)*	*Value Diff. AA and BIH (1)*	*Relation (1)/(2)*
A	1,00	0,25	25,0%
K	1,00	0,50	50,0%
L	3,00	0,50	16,7%
E	7,00	3,50	50,0%
G	7,00	3,50	50,0%
D	15,00	5,00	33,3%
C	27,00	7,50	27,8%
J	34,00	0,50	1,5%
B	63,00	13,50	21,4%
H	79,00	6,00	7,6%
I	102,00	14,50	14,2%
F	125,00	12,00	9,6%

Table 5 - Analysis of Results from Algorithms

To know if this deduction could be generalized, range amplitude (taking *ub* - *lb*) and the difference (absolute value) between the result value for AA and HBI were calculated for all runs, along with the relation (in percent) among these two numbers. That information is contained in Table 5, presented sorted in ascending order by the second column (amplitude of result range).

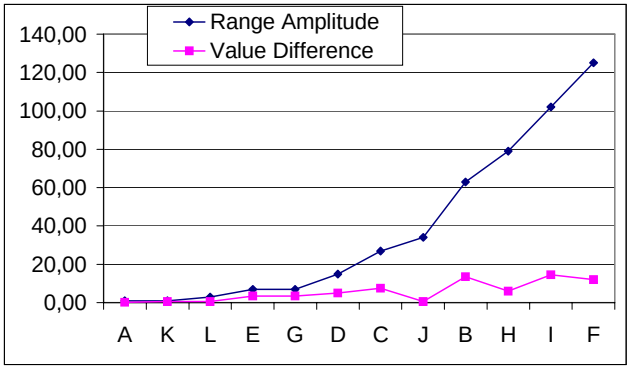

Fig. 9 - Comparison of Results

From Table 5 a graph was made (Fig. 9), where the tendency is confirmed: the broader the amplitude of the result range in BIH, greater the difference between the result value of that algorithm and the answer from AA. It can be inferred that it is convenient to apply *Brooks-Iyengar Hybrid* in case of using sensors with a large precision range.

6. CURRENT APPLICATIONS

The present section is mainly devoted to show several applications that have been developed using this Operating System, and a new set of programs being built at present.

I. Supervisory Control And Data Acquisition (SCADA)

The first developed application was a SCADA program developed with academic purposes. It was written previously to run under MINIX and later adapted to execute in a real-time environment. The SCADA application is built to be used to supervise a set of industrial processes. Different parameters can be defined for each process, including ports to be read, values to be recorded and alarms to be raised. Data acquired by the program can also be monitored from another computer through the serial ports. A history log file is generated, allowing the revision (and printout) of the activity that occurred during program execution.

A SCADA tool is a good application to test RT-MINIX with real processing conditions. It is composed of several periodic and sporadic real-time tasks running concurrently. It also includes a set of soft real-time tasks combined with interactive processes.

J. Model of a Bottle-filling Line

A prototype of a bottle-filling system (as described in [19]) is currently under construction, with the aim of using RT-MINIX as the RTOS to control such a real process.

The proposed system modeled in that work consists in a number of bottle-filling lines fed by a single vat containing the liquid to be bottled. The bottle size may differ from line to line. The tasks of the control system are to control the level, the pH and the temperature of the liquid in the vat, to manage the movement and filling of bottles in the various lines, and to exchange and log information with human operators working with the individual lines and a supervisor monitoring the entire system.

With several concurrent tasks (both periodic and aperiodic), this prototype will impose RT-MINIX real-world constraints to play with.

7. PRESENT WORK

The sensor integration problem and tolerance of failures from replicated (redundant) sensors can now be studied in depth with help of RT-MINIX thanks to the incorporated sensing algorithms. A possible work line is deal with multidimensional sensors (replacing each interval corresponding to a physical value by a vector of intervals).

The algorithms presented in Section 2 are only two examples of a long and growing list of scheduling algorithms. Real-time guarantees in the presence of faults along with fault tolerant scheduling strategies are very interesting fields to extend the present state of RT-MINIX. Feasible Shortest Path (FSP) and Linear Time Heuristic (LTH) are models that can be studied and compared, with a future implementation in RT-MINIX depending on results to be obtained.

One of the problems associated with scheduling algorithms is priority inversion. [20] presents a very clear example to definitely understand priority inversion, a case that occurred during the NASA Mars Pathfinder mission in 1997.

Any task within RT-MINIX can have a priority: if new scheduling algorithms to be implemented will consider that value to pick a task instead of another one, care must be taken in order to handle this characteristic properly. It is possible that a task with medium priority be scheduled while a high priority task is waiting for a resource that is blocked by a low priority task. A solution to that dilemma known as priority inheritance was identify and proposed in [21]. Tasks should inherit the right value to avoid priority inversion and furthermore deadline missing, thus improving the overall performance of the scheduling algorithms.

8. 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.

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.

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.

K. Future Work

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-tolerant environment.

9. 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.

10. 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] B. Srinivasan. KURT: The KU Real-Time Linux. Online at http://hegel.it.ukans.edu/projects/kurt

[4] H. Tokuda, T. Nakajima, P. Rao. Real-time MACH: Towards a predictable real-time system. *Proceedings of USENIX MACH Workshop*, volume 1, 1990.

[5] M. Barabanov, V. Yodaiken. RT-Linux: A Real-Time UNIX. Online at http://luz.cs.nmt.edu/pub/rtlinux

[6] J. Stankovic, K. Ramamrithman. The design of the Spring kernel. In Proc. of 8th RealTime Systems Symposium. 1991.
[7] M. Saksena, J da Silva, A. Agrawala. Principles of Real-Time Systems, chapter Design and Implementation of Maruti. Prentice-Hall, 1994.
[8] K. Jeffay, D. Stone, D. Poitier. Kernel support for efficient, predictable real-time systems. Proceedings of the IEEE Workshop on RTOS, pp. 8-31, 1991.
[9] LynxOS – Hard Real-time OS Features and Capabilities, online at http://www.lynx.com/products/ds_lynxos.html
[10] QNX Realtime OS, online at http://www.qnx.com/products/qnxrtos.htm
[11] G. Wainer, "Implementing Real-Time Scheduling in a Time-Sharing Operating System", *ACM Operating Systems Review*, July 1995.
[12] 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.
[13] D. Polakoff, P. Rogina, W. Ruaro, E. Szulsztein, G. Wainer, "Real-time modifications of the Minix Operating System" (in Spanish), Internal Report, CS Dept., FCEyN, UBA, December 1997.
[14] 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
[15] N. Wolowick, M. Cuenca Acuña, G. Wainer, "Joining the scheduling queues in Minix Operating System" (in Spanish), Internal Report, CS Dept., FCEyN, UBA, July 1998.
[16] R. Brooks, S. Iyengar, "Robust Distributed Computing and Sensing Algorithm", *IEEE Computer*, pp 53-60, June 1996.
[17] K. Marzullo, "Tolerating failures of continuous-valued sensors", *ACM Transactions on Computer Systems*, 8(4):284-304, November 1990.
[18] D. Jayasimha, "Fault Tolerance in a Multisensor Environment", Dept. of Computer Science, The Ohio University, May 1994.
[19] P. Ward, S. Mellor, "Structured Development for Real-Time Systems", Appendix B, Yourdon Press, 1985.
[20] M. Jones, What happened on Mars?, document available on line at http://www.cs.cmu.edu/afs/proejct/art-6/www/mars.html
[21] L. Sha, R. Rajkumar, J. Lehoczky, Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Trans. on Comp.*, 39:1175-1185, September 1990.