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

TESIS DE LICENCIATURA

Tolerancia a Fallas en Sistemas de Tiempo Real

Pablo J. Rogina
L.U. 746/92
pr6a@dc.uba.ar

Director: Dr. Gabriel Wainer

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

http://www.dc.uba.ar

Diciembre 1999

Tolerancia a Fallas en Sistemas de Tiempo Real

Los sistemas de computación ya se encuentran inmersos en prácticamente todas las actividades humanas. En particular, los sistemas de tiempo real están presentes en tareas cada vez más complejas y donde un error puede conducir a situaciones catastróficas (incluso con peligro para vidas humanas). Por eso, las capacidades de tolerancia a fallas de este tipo de sistemas son críticas para su éxito a lo largo de su ciclo de vida. Si bien las estrategias de tolerancia a fallas son desarrolladas desde hace tiempo, su orientación principal fueron los sistemas distribuidos. Distintas características de los sistemas de tiempo real deben ser convertidas para hacerlas tolerantes a fallas, por lo cual hay un gran campo de investigación y desarrollo.

Este trabajo introduce primero la terminología apropiada sobre sistemas de tiempo real y tolerancia a fallas por separado, para luego presentar una revisión de los estudios de tolerancia a fallas aplicables específicamente a sistemas de tiempo real (en distintas áreas). Finalmente, algunos de esos conceptos se prueban en un modelo real, para estar en condiciones de extraer las conclusiones correspondientes y comentar puntos de interés para trabajos futuros.

Fault Tolerance in Real-Time Systems

Computing systems are already among almost any human activities. In particular, real-time systems 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 of this kind of systems are critical to their success during their lifetime cycle. Although fault tolerance strategies are developed since a long time ago, they were oriented mainly to distributed systems. Distinct features of real-time systems must be converted to become fault tolerant, thus bringing a great field for research and development.

This work first presents proper terminology about real-time systems and fault tolerance separately, and then brings a revision of studies about fault tolerance items specific to real-time systems (different subjects). Finally, selected aspects from the ones presented earlier are proven in a real model, in order to take the corresponding conclusions and to discuss points of interest for future works.

Contenido

Figuras

Programas

Tablas

1. Introducción

"Si un hombre se imagina una cosa, otro la tornará en realidad"

Julio Verne

Propósito

En este trabajo se describen las experiencias realizadas al comprobar en forma práctica ciertas características de tolerancia a fallas aplicables a sistemas de tiempo real. Como las técnicas disponibles son diversas y las áreas de aplicación son variadas, se optó por desarrollar un modelo real donde aplicar replicación de sensores y emplear algoritmos de sensado robusto distribuido.

Para llevar a cabo estas pruebas era necesario contar con un sistema operativo de tiempo real. La elección recayó en RT-MINIX, el cual fue modificado y extendido para disponer de servicios nuevos que tenían aplicación directa con el tema a desarrollar. También se construyó un modelo de sensores replicados (en la forma de potenciómetros lineales) para proveer de valores variables a los algoritmos de sensado que se fueran a implementar.

La nueva versión de RT-MINIX incorpora un nuevo controlador de dispositivos para la palanca de juegos (*joystick*) escrito desde cero; necesario para poder conectar sensores analógicos y acceder al conversor A/D incorporado en las PCs. Se modificó además el manejo de colas de tareas: reducir la cantidad de colas mejoró el tiempo de respuesta y simplificó el trabajo de planificación, una acción tendiente hacia la tolerancia a fallas del sistema operativo. Los algoritmos de sensado robusto seleccionados primero se codificaron y probaron como programas de usuario sobre RT-MINIX; una vez que se comprobaron los resultados, siguió la tarea de incluirlos como un servicio dentro del núcleo del sistema operativo.

Este documento presenta en el capítulo 2 los conceptos relacionados con sistemas de tiempo real por un lado, y una detallada descripción de términos, técnicas y características usadas en el área de tolerancia a fallas en general por otro; sirve a los efectos de introducir los fundamentos necesarios al propósito de esta tesis; también se brindan referencias para profundizar cada uno de dichos temas. Dentro del capítulo 3 se desarrolla el estado del arte en cuanto al avance de la tolerancia a fallas específicamente aplicada en sistemas de tiempo real. Para ello se muestran las técnicas disponibles en varias áreas: especificación y diseño, algoritmos de planificación y trabajos con sensores y actuadores. Las experiencias con el modelo real están incluidas en el capítulo 4: se describen los alcances de los trabajos realizados en cada tema para lograr la nueva versión de RT-MINIX. En el capítulo 5 están los detalles de implementación; se enumeran los archivos nuevos y modificados, instrucciones para configurar y compilar el sistema operativo con distintas opciones y se comentan varios programas de ejemplo. El capítulo 6 contiene las conclusiones alcanzadas en el marco del trabajo llevado a cabo, junto a una lista de tareas que pudieran realizarse a futuro. Para finalizar, la bibliografía consultada y referencias usadas en la preparación de este trabajo se encuentran listadas en el capítulo 7.

2. Conceptos

"No con la edad, sino con el ingenio se adquiere la sabiduría"

Plauto

Sistemas de Tiempo Real

Introducción

Los sistemas de tiempo real son aquellos que no sólo tienen que producir resultados correctos, sino que tienen que realizarlos en un momento determinado. La corrección del sistema depende del resultado del cómputo *y* del momento en que los resultados se producen. También podemos pensarlos como sistemas cuyo comportamiento está dado no sólo por la sucesión de acciones que realizan, sino por el tiempo de ejecución y el momento en que las mismas se realizan.

Los sistemas de tiempo real se asocian en general con aplicaciones de control y comando, y el número de computadoras que controlan procesos en el mundo real está en continuo crecimiento. Con ejemplos que se extienden desde sistemas de manufactura flexibles y cadenas de producción, señalización de ferrocarriles; hasta control de vuelo de aeronaves y vehículos espaciales, experimentos en laboratorios, pasando por sistemas de control de procesos (especialmente en entornos peligrosos, como plantas químicas).

El número de aplicaciones donde las restricciones de tiempo son críticas es cada vez mayor, y el cumplimiento de las metas de tiempo impuestas a estos sistemas es muy importante, por el amplio dominio de este tipo de sistemas: control de tráfico aéreo para un país completo, autopistas inteligentes, o sistemas de comunicaciones multimedia de alta velocidad, por citar algunos casos. Los sistemas de tiempo real tienen que considerarse como una tecnología clave (*key enabling technology*) [Sta96]. Otras características de los sistemas de tiempo real es la comunicación con el mundo físico, en lugar de hacerlo principalmente con un

operador (o usuario) humano[1]; y la supeditación a los eventos externos, los cuales fijan verdaderamente el ritmo de ejecución de las acciones.

Los ejemplos citados anteriormente dan una idea de la importancia de la tolerancia a fallas de un sistema de tiempo real. Deben ser sumamente confiables y soportar la pérdida de partes de hardware o errores en el software para poder continuar con las funciones más críticas. Las etapas de especificación y diseño son importantes a la hora de incorporar capacidades de tolerancia a fallas. Como en cualquier otro tipo de sistemas, especificaciones incompletas o erróneas pueden ser muy caras, cuando el sistema no sea capaz de ejecutar su carga crítica en un tiempo máximo.

Componentes

Como una aproximación simple, los sistemas de tiempo real consisten en un sistema de control y un sistema controlado.

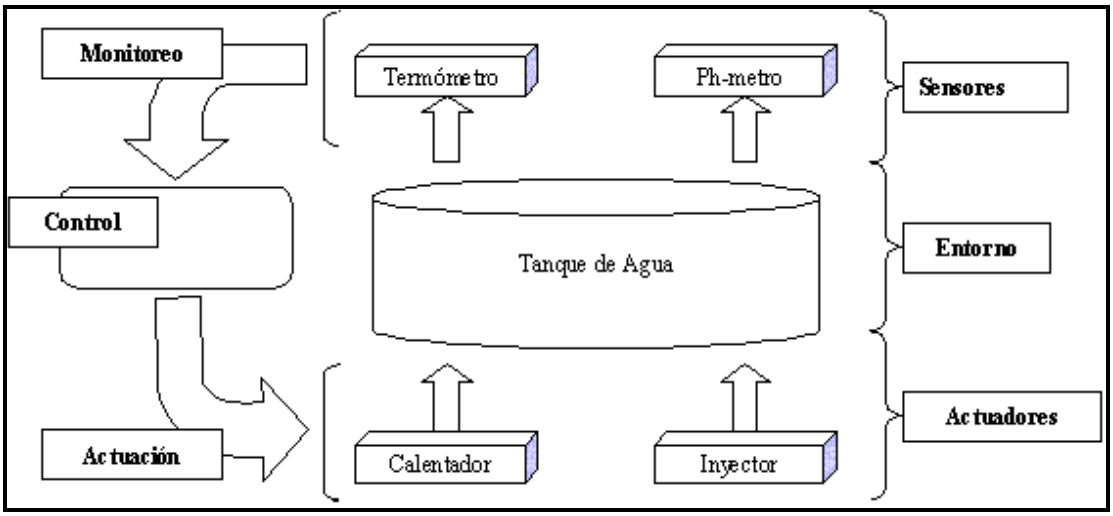

Figura 1 - Esquema de un Proceso de Control

Tomemos un sencillo proceso de manufactura: un tanque lleno de agua que debe mantenerse a una temperatura establecida con un pH controlado (Figura 1). El entorno (o ambiente) es el sistema controlado con el cual interactua la

[1] Aunque el operador puede ser informado del estado del proceso

computadora. Mantener la temperatura o balancear el ph son las *tareas* a llevar a cabo por este sistema de tiempo real. Otra característica de los sistemas de tiempo real es que sus tareas tienen restricciones de tiempo (conocidas como *metas*) que deben cumplirse o pueden provocar consecuencias catastróficas (dejar que el contenido del tanque se vuelva ácido arruinaría el proceso y sería una gran pérdida económica en este caso). Por lo tanto, es necesario que el sistema controlante monitoree el entorno. De este modo, se hacen necesarios sensores (para el ejemplo, un termómetro y un medidor de ph) y son componentes que aparecen siempre en sistemas de tiempo real. Por razones de seguridad, es imperioso revisar el entorno en forma periódica. Y ello conduce a llevar a cabo acciones de rutina para controlar la situación. El sistema controlante cambia el entorno por medio de otros componentes también siempre presentes en los sistemas de tiempo real: actuadores (volviendo al ejemplo, un calentador y un inyector de ácido). Un estudio detallado de los conceptos y avances específicos de sistemas de tiempo real está más allá del alcance de este trabajo, pero se presenta en [Sta92a].

El proceso de manufactura propuesto necesita, para su correcta operación, de las siguientes funciones:

- Sensado: debe conocerse el estado actual del mundo real (medición de la temperatura del agua y su valor de acidez)
- Control y Cálculo: deben controlarse los valores del mundo real. La temperatura debe estar dentro de un determinado rango para el proceso, como así también la acidez.
- Actuación: debe cambiarse el estado actual del mundo real. Mantener la temperatura accionando el calentador el tiempo necesario.

Tanto el monitoreo como la actuación, se llevan a cabo por medio de dispositivos de interfaz, que incluyen conversores AD/DA (Analógico-Digital/ Digital-Analógico), líneas de entrada/salida (E/S) digital o generadores de pulsos. Cada tipo necesita de un software de control (conocido como *drivers* de E/S) para su apropiado funcionamiento.

Los elementos mencionados, sensores y actuadores (hardware) y las tareas (software), son fuentes potenciales de fallas dentro de un sistema de tiempo real. En general, la tolerancia a fallas del hardware es una disciplina encarada por la Ingeniería Electrónica, mientras que la tolerancia a fallas en el software es una empresa de las Ciencias de la Computación.

Las tareas en un sistema de tiempo real, para realizar correctamente su trabajo, deben ser sincronizadas en forma apropiada, a través de una planificación cuidadosa. Esto a su vez, está acotado, porque el orden de las tareas viene dado en general por la ocurrencia de eventos externos del mundo real. El planificador de tareas debe ser altamente dinámico. Aparece aquí otro componente importante de un sistema de tiempo real: la planificación. Su objetivo es brindar tiempos de respuesta predecibles para la mayor cantidad posible de tareas. La característica de planificación de un sistema de tiempo real es el grado de utilización de recursos en el cual puede garantizarse los requisitos de tiempo de las tareas.

El estudio de distintos métodos de planificación (*scheduling*) dentro de los sistemas de tiempo real es una rama con gran difusión. Mucho trabajo se ha realizado hasta el presente y han surgido diversos enfoques sobre este tema. Nuevamente, la planificación de tareas es un elemento que debe ser tolerante a fallas, medido especialmente en que sea predecible y garantice el cumplimiento de las metas de las tareas.

Tolerancia a Fallas

Introducción

Decimos que un sistema *falla* cuando no cumple con su especificación. Dependiendo de la complejidad e importancia del sistema, esta falla puede tolerarse (datos estadísticos erróneos en un censo, que pueden calcularse nuevamente en otra ocasión) o pueden llevar a una catástrofe (un sistema de control de tráfico aéreo). El uso cada vez mayor de computadoras en misiones donde la seguridad es crítica, hizo necesario que la capacidad de evitar y tolerar fallas se incremente día a día.

¿Qué es una falla? Es un error, causado quizás por un problema de diseño, construcción, programación, un daño físico, uso, condiciones ambientales adversas o un error humano. De este modo, las fallas pueden aparecer tanto en el hardware como en el software. La falla de un componente del sistema no conduce directamente a la falla del sistema, pero puede ser el comienzo de una serie de fallas que quizás sí terminen con la falla del sistema.

[Smi88] explica como la tolerancia a fallas de hardware es una disciplina precisa, opuesta a la tolerancia a fallas de software, de la cual el autor dice “retiene un aire de alquimia”. Hay que marcar la diferencia respecto de garantizar que los componentes individuales sean muy confiables, dado que eso asume que el sistema falle si alguno de los componentes lo hace. Respecto de este punto, el trabajo tradicional (y en el cual no haremos hincapié) se ha centrado en el estudio estadístico de las fallas de los componentes electrónicos. La industria ha avanzado muchísimo en este campo y la tecnología de componentes es cada vez más confiable. Los valores de MTBF (*Mean Time Between Faults* - Tiempo Medio Entre Fallas) han crecido en forma constante durante los últimos tiempos. Basta recordar que se asumía como normal cierto número de sectores defectuosos en los discos rígidos usados hace 6 o 7 años atrás, algo totalmente inaceptable hoy en día.

El objetivo al diseñar y construir un sistema tolerante a fallas consiste en garantizar que continúe funcionando de manera correcta como un todo, incluso ante la presencia de fallas. La idea es que el sistema pueda seguir adelante (sobrevivir) a las fallas de los componentes, en lugar de que éstas sean poco probables. Mucha de la bibliografía disponible sobre tolerancia a fallas se refiere especialmente a la forma en que se trata el tema en sistemas distribuidos. Por la naturaleza de esos sistemas (la distribución) surgen entonces distintas áreas donde superar las fallas: los procesadores, las comunicaciones y los datos.

Redundancia

El método general para la tolerancia de fallas es el uso de redundancia. Hay tres tipos posibles de redundancia:

De información podemos agregar código de Hamming para transmitir los datos y recuperarse del ruido en la línea por ejemplo. También en sistemas distribuidos, surge la replicación de datos. Esto trae aparejado varios problemas, ya que administrar los datos replicados no es fácil, las soluciones simplistas no funcionan, y hay que pagar un precio por el acceso y disponibilidad de los datos. No vamos a ahondar en este tema, que es complejo y representa un caso de estudio en sí mismo

Del tiempo aquí se realiza una acción, y de ser necesario, se vuelve a realizar. Es de particular utilidad cuando las fallas son transitorias o intermitentes.

Física se agrega equipo adicional para permitir que el sistema tolere la pérdida o mal funcionamiento de algunos componentes.

Esto da lugar a dos formas de organizar los equipos redundantes: la activa y el respaldo primario. Para el primer caso, todos los equipos funcionan en paralelo para ocultar la falla de alguno(s) de ellos. Por su parte, el otro esquema utiliza el equipo redundante de respaldo, sólo cuando el equipo principal falla.

La vida real nos da ejemplos concretos de réplica activa para tolerar fallas mediante redundancia física. Los mamíferos tienen dos oídos, dos pulmones, dos riñones, etc.; en aeronáutica aviones con cuatro motores pueden volar con tres; deportes con varios árbitros, por si alguno omite un evento. Sin embargo, este esquema trae aparejada la necesidad de establecer protocolos de votación. Supongamos (para llevarlo al terreno de STR) tres sensores de presión. Si dos o tres de los valores sensados son iguales, el valor usado es esa entrada. Aparecen problemas si los tres valores son distintos. Hay dos enfoques a la hora de construir protocolos de votación: optimistas y pesimistas. También habrá que considerar que pasa si falla el algoritmo encargado de administrar la votación.

La réplica de respaldo también se manifiesta en el mundo real: el gobierno con el vicepresidente; la aviación, con el copiloto; los automóviles, con las ruedas de auxilio. Este esquema aparece como una solución más sencilla pues no hay necesidad de votación, y además se requieren menos equipos redundantes (en el caso más simple, un primario y un respaldo). Pero tiene la desventaja de trabajar mal ante fallas bizantinas puesto que el primario no da señales claras de fallar.
Para cualesquiera de los métodos a emplear se deben considerar:

- el grado de replicación a usar
- el desempeño en el caso promedio y el peor caso, en ausencia de fallas
- el desempeño en el caso promedio y el peor caso, en presencia de fallas

Conceptos y Terminología

A lo largo de esta sección se introducirá un marco conceptual para tolerancia a fallas. La idea es presentar todos los elementos necesarios para estar en condiciones de analizar luego dichas técnicas y estrategias en su aplicación concreta en los sistemas de tiempo real. Un trabajo muy importante cuyo interés fue brindar definiciones informales pero precisas sobre los distintos atributos que caracterizan la confiabilidad de sistemas de computación es [Lap85]. En él encontramos una propuesta de taxonomía, mostrada en la Figura 2 (y citada además en [Smi88]). Veremos que el nivel más abstracto es la *confiabilidad*. Este

requisito se cumple sabiendo qué hace a un sistema 'no confiable'. Son los *deterioros*. Los *medios* son los métodos sobre los cuales se construye un sistema confiable, mientras que las *medidas* son los criterios para evaluar el éxito de nuestra tarea.

Un sistema existe en un entorno (una cápsula espacial en el espacio exterior), y tiene operadores y usuarios (en ciertos casos, posiblemente los mismos). El sistema provee respuestas al operador y servicios al usuario. Los operadores se consideran dentro del sistema porque generalmente los procedimientos del operador son parte del diseño del sistema y muchas funciones, incluyendo la recuperación de fallas, pueden involucrar acciones del operador. Los sistemas se desarrollan para satisfacer un conjunto de requisitos que cumplen una necesidad. Un requisito que es muy importante en algunos sistemas es que sean altamente confiables. La tolerancia a fallas es un medio de conseguir esa confiabilidad.

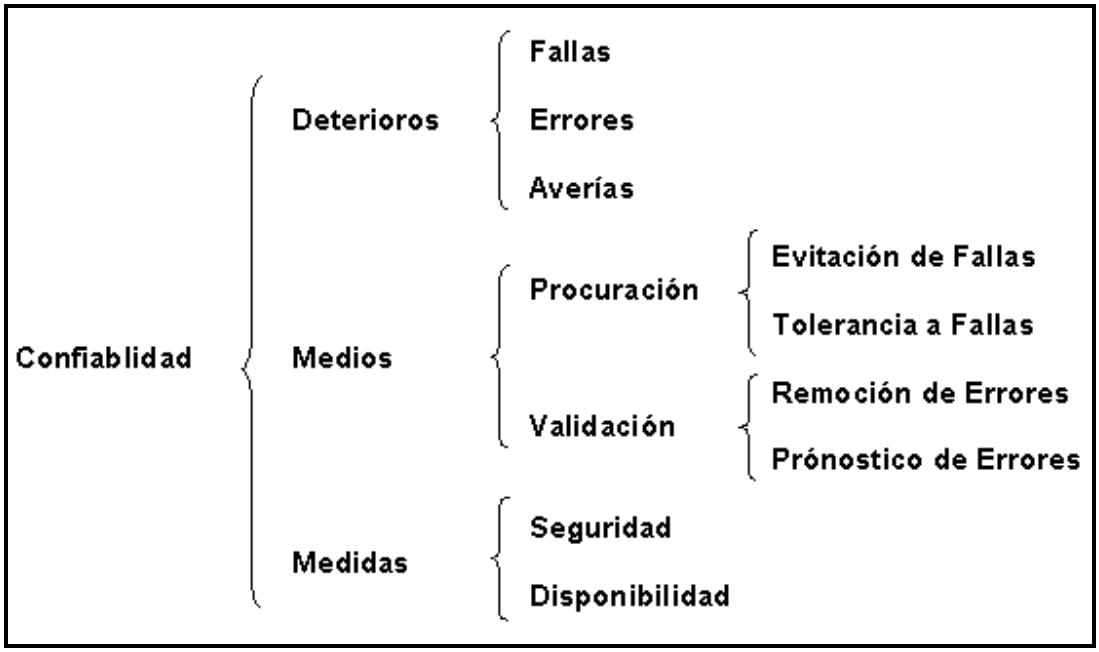

Figura 2 - Taxonomía de Laprie

Hay tres niveles en los cuales puede aplicarse tolerancia a fallas. Tradicionalmente, la tolerancia a fallas se ha usado para compensar fallas en recursos de computación (hardware). Al manejar recursos de hardware

adicionales, el subsistema de computación aumenta su habilidad de continuar operando. Las medidas para *tolerancia a fallas de hardware* incluyen comunicaciones redundantes, procesadores replicados, memoria adicional y fuentes de energía redundantes. La tolerancia a fallas de hardware fue importante en los primeros tiempos de la computación, cuando el tiempo medio entre fallas (MTBF - *Mean Time Between Faults*) se medía en minutos.

Un segundo nivel de tolerancia a fallas reconoce que una plataforma de hardware tolerante a fallas no garantiza, por sí misma, alta disponibilidad al usuario del sistema. Todavía es importante estructurar el software para compensar fallas tales como cambios en programas o estructuras de datos debidas a errores transitorios o de diseño. Esto es *tolerancia a fallas de software*. En este nivel se usan a menudo mecanismos como puntos de control/reinicio, bloques de recuperación y programas de múltiples versiones.

En un tercer nivel, el subsistema de computación puede proveer funciones para compensar fallas en otros componentes del sistema que no están basados en computadoras. Esto es la *tolerancia a fallas del sistema*. Aquí se ven medidas específicas de la aplicación.

Sistemas Confiables

Los peligros a un sistema son una realidad de la vida. También lo son las fallas. A pesar de eso, queremos que nuestros sistemas sean confiables. Para que un sistema sea confiable, debe estar disponible (es decir, listo para usarse cuando se necesite), ser digno de confianza (esto es, capaz de proveer continuidad de servicio mientras se está usando), seguro (es decir, no tener una consecuencia catastrófica al entorno) y fuerte (es decir, capaz de preservar la confidencialidad).

Aunque estos atributos del sistema pueden considerarse en forma aislada, de hecho son interdependientes. Un sistema que no es confiable también no está disponible (al menos cuando no está operando correctamente). Un sistema seguro que no permite el acceso autorizado también no está disponible. Un sistema no

confiable para controlar reactores nucleares probablemente no sea uno seguro tampoco.

Especificaciones de Confiabilidad

El grado de tolerancia a fallas que un sistema requiere, puede especificarse cuantitativa o cualitativamente.

Objetivos Cuantitativos

Un objetivo de confiabilidad cuantitativa a menudo se expresa como la relación de fallas máxima permitida. Por ejemplo, el valor que se usa generalmente en sistemas de computación para aviones comerciales es menor a 10^{-9} fallas por hora. El problema al establecer los requisitos de esta manera es que es difícil conocer cuando se han alcanzado. Butler ha señalado que los métodos estadísticos estándar no pueden usarse para mostrar la confiabilidad de software, tanto normal como tolerante a fallas. También está claro que no hay forma de lograr la confianza de que un sistema cumpla tal objetivo de confiabilidad usando pruebas al azar (*random testing*). A pesar de eso, los objetivos de confiabilidad a menudo se expresan de esta manera.

Objetivos Cualitativos

Un método alternativo de especificar características de confiabilidad de un sistema es especificarla cualitativamente. Especificaciones típicas incluirían:

Fail-safe	Diseñar el sistema de modo que, cuando aguanta un número especificado de fallas, falle en un modo seguro. Por ejemplo, un sistema de señalización ferroviario, se diseña para que al fallar, todos los trenes se detengan.
Fail-op	Diseñar el sistema de modo que, cuando aguanta un número especificado de fallas, todavía provee un subconjunto de su comportamiento especificado.
Ningún punto de falla	Diseñar el sistema de modo que la falla de cualquier componente simple no cause la falla del sistema. Tales

sistemas se diseñan de modo que el componente que ha fallado pueda reemplazarse antes de que ocurra otra falla.

CONSISTENCIA

Diseñar el sistema de modo que toda la información suministrada por el sistema es equivalente a la información suministrada por una instancia sin fallas del sistema.

Fallas y Averías

Los términos falla y falta a menudo se usan mal. Una describe las situaciones a ser evitadas, mientras la otra describe el problema a ser evitado. Con el tiempo, avería ha llegado a ser definida en términos de servicios especificados provistos por un sistema. Esto evita definiciones circulares que involucren términos sinónimos como ser defecto, etc. Se dice que un sistema tiene una *avería* si el servicio que provee al usuario se desvía del cumplimiento con la especificación del sistema por un período de tiempo especificado. Mientras puede ser difícil llegar a una especificación no ambigua del servicio a ser provisto por cualquier sistema, el concepto de una especificación acordada es la más razonable de las opciones para definir servicio satisfactorio y la ausencia de servicio satisfactorio, avería.

La definición de avería como la desviación del servicio provisto por un sistema de la especificación del sistema elimina errores o fallas de “especificación”. Mientras esta aproximación puede estar evitando el problema al definirlo de todos modos, es importante tener alguna referencia para la definición de avería, y la especificación es una elección lógica. La especificación puede considerarse como un límite a la región de interés del sistema, que se discute luego. Es importante reconocer que cada sistema tiene una especificación explícita, la cual está escrita, y una especificación implícita que el sistema debería al menos comportarse tan bien como una persona razonable podría esperar basada en experiencias con sistemas similares y con el mundo en general. Claramente, es importante hacer la especificación tan explícita como sea posible.

Se ha convertido en una práctica definir fallas en términos de avería. El concepto más cercano al entendimiento común de la palabra falla es una que define a una *falla* como la causa adjudicada de una avería. Esto encaja con una aplicación común del verbo fallar, que involucra determinar causa o culpa anexa (*affixing blame*). Sin embargo, esto requiere entender de como se producen las averías. Una forma alternativa de ver a las averías es considerarlas fallas en otros sistemas que interactuan con el sistema bajo consideración, tanto un sistema interno al sistema considerado, un componente del sistema considerado, o un sistema externo que interactua con el sistema considerado (el entorno). En el primer caso, la conexión entre averías y fallas es causa; en el segundo es nivel de abstracción o ubicación.

Las ventajas de definir averías como fallas de componentes/sistemas interactuantes son:

1. Uno puede considerar fallas sin la necesidad de establecer una conexión directa con una avería, así se puede tratar fallas que no causan averías, es decir, el sistema es naturalmente tolerante a fallas.
2. La definición de una falla es la misma que la de una avería con la única diferencia de los límites del sistema o subsistema relevante.

Esto significa que podemos considerar a un defecto interno obvio como una falla sin tener que establecer una relación causal entre el defecto y una avería en el límite del sistema.

Con lo antes mencionado, una *falla* se definirá como la avería de (1) un componente del sistema, (2) un subsistema del sistema, o (3) otro sistema que ha interactuado o está interactuando con el sistema considerado. Cada falla es una avería desde algún punto de vista. Una avería puede conducir a otras fallas, o a otra avería, o a nada.

Un sistema con fallas que puede continuar proveyendo su servicio, no falla. Tal sistema se dice que es *tolerante a fallas*. De este modo, una motivación importante para diferenciar entre averías y fallas es la necesidad de describir la

tolerancia a fallas de un sistema. Un observador que inspeccione los componentes internos de un sistema podría decir que el componente defectuoso ha fallado, porque el punto de vista del observador no está a un nivel de detalle más bajo. El efecto observable de una falla en el límite del sistema se llama *síntoma*. El síntoma más extremo de una falla es una avería, pero también podría ser algo tan benigno como una lectura alta en un termómetro. Los síntomas se discutirán luego en detalle.

Consideremos un sistema de computación que corra un programa de control de la temperatura de una caldera al calcular la llama del quemador para la caldera. Si un bit de memoria se pone en uno, es una falla. Si la falla de la memoria afecta la operación del programa de forma tal que la salida del sistema de computación hace que la temperatura de la caldera se eleve fuera de la zona normal, es una avería en el sistema de computación y una falla en el sistema general de la caldera. Si hay un termómetro mostrando la temperatura de la caldera, y su aguja se mueve hacia la zona “amarilla” (anormal pero aceptable) eso es un síntoma de la falla del sistema. Por otra parte, si la caldera explota por el cálculo defectuoso de la llama, eso es una avería (catastrófica) del sistema.
Las razones para la falla de la memoria son múltiples. El chip usado podría haber sido fabricado según la especificación (falla de fabricación), el diseño de hardware podría haber causado que se aplicara demasiado voltaje al chip (falla de diseño del sistema), el diseño del chip puede hacer propensas tales fallas (falla de diseño del chip), un ingeniero puede haber invertido inadvertidamente dos cables al realizar mantenimiento (falla de mantenimiento).

Relaciones de Dependencia

Un interés en el diseño de sistemas tolerantes a fallas es la identificación de dependencias. Las dependencias pueden ser estáticas, siendo las mismas permanentes durante toda la vida del sistema, o pueden cambiar, tanto por diseño o por efecto de las fallas. Se dice que un componente de un sistema depende de otro componente si la corrección del comportamiento del primer componente requiere la operación correcta del segundo componente. Tradicionalmente, se

considera que las posibles dependencias en un sistema forman un grafo acíclico. El término análisis del árbol de fallas parece implicar ésto, entre otras cosas. En realidad muchos sistemas exhiben este comportamiento, en el cual una falla conduce a otra la cual conduce otra hasta que eventualmente se produce una avería. Es posible, sin embargo, para una relación de dependencia ciclar nuevamente hasta ella. Se dice que una relación de dependencia es acíclica si forma parte de un árbol. Una relación de dependencia cíclica es aquella que no puede describirse como parte de un árbol, sino más bien como parte de un grafo cíclico dirigido.

Regiones de Fallas

Definir una región de avería limita la consideración de fallas y averías a una porción del sistema y su entorno. Esto es necesario para asegurar que los esfuerzos de especificación del sistema, análisis y diseño se concentran en porciones de un sistema que puede ser observado y controlado por el diseñador y el usuario. Ayuda a simplificar una tarea de otro modo abrumadora.
Un sistema típicamente está hecho con gran cantidad de partes componentes. Estos componentes están, a su vez, hechos con subcomponentes. Esto continua arbitrariamente hasta que se llega a un componente *atómico* (un componente que no es divisible o que hemos elegido no dividir en subcomponentes). Aunque teóricamente todos los componentes son capaces de tener fallas, para cualquier sistema hay un nivel más allá del cual las fallas “no son de interés”. Este nivel se llama *piso de fallas*. Los componentes atómicos yacen en el piso de fallas. Nos interesamos en fallas que surgen en componentes atómicos, pero no en fallas que yacen dentro de estos componentes.

De manera similar, como los componentes se agregan en un sistema, eventualmente el sistema está completo. Todo los demás (esto es, el usuario, el entorno, etc.) no es parte del sistema. Este es el límite del sistema. Las averías ocurren cuando las fallas alcanzan los límites del sistema.

Como se muestra en la Figura 3, el *espacio de interés* comienza en los límites entre el sistema y el usuario y entre el sistema y el entorno, y termina en el piso de fallas. Las fallas debajo de este piso no son distinguibles, tanto porque no son completamente entendidas o porque son demasiado numerosas. De manera informal, el espacio de interés es el área en la cual las fallas son de interés.

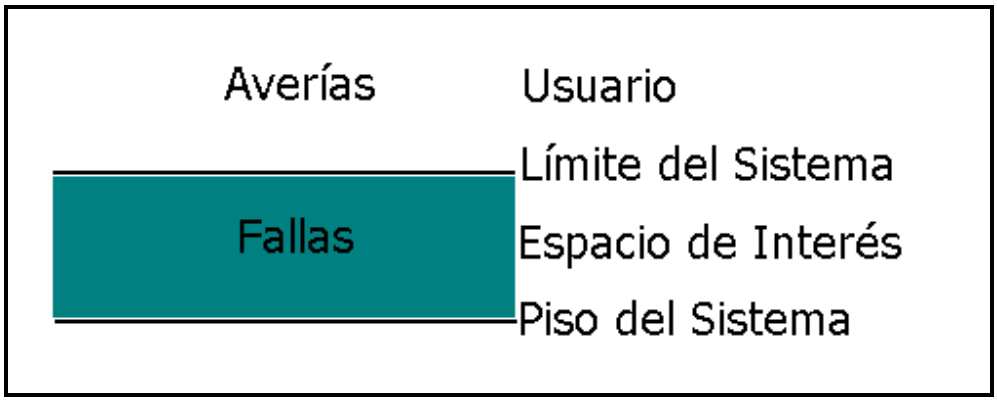

Figura 3 - Región de Fallas

Clases de Fallas

Ningún sistema puede hacerse para que resista todas las fallas posibles, de modo que es esencial que las fallas se consideren a través del proceso de definición de los requisitos y diseño del sistema. Sin embargo, no es práctico enumerar todas las fallas a ser toleradas; las fallas deben agruparse en clases de fallas manejables.

Las fallas pueden clasificarse basadas en la localidad (componente atómico, componente compuesto, sistema, operador, entorno), en efecto (tiempo, datos) o en causa (diseño, daño). Otros criterios posibles de clasificación incluyen duración (transitorias, persistentes) y efecto sobre el estado del sistema (ruptura, amnesia, amnesia parcial, etc.).

Ya que la ubicación de una falla es tan importante, este criterio es un punto de partida lógico al clasificar fallas.

Localidad

Fallas de Componentes Atómicos

Una *falla de componentes atómicos* es una falla en el piso de fallas, esto es, un componente que no puede dividirse para propósitos de análisis.

En un sistema de computación, las fallas de substratos pueden aparecer en diversas formas. Por ejemplo, una falla en un bit de memoria no es una falla atómica si los detalles de la memoria están fuera del alcance de interés. Una falla así, podría o no aparecer como una falla de memoria, dependiendo de la habilidad de la memoria para enmascarar las fallas de bits.

Fallas de Componentes Compuestos

La *falla de un componente compuesto* es aquella que aparece dentro de un conjunto de componentes atómicos mas que en un componente atómico. Puede ser el resultado de una o más fallas de componentes atómicos. La avería de una unidad de disco en un sistema de computación es un ejemplo de avería de componente compuesto.

Fallas a Nivel de Sistema

Una *falla a nivel de sistema* es aquella que aparece en la estructura de un sistema más que en sus componentes. Tales fallas generalmente son interacción o integración de fallas, es decir, ocurren debido a la forma en que el sistema está armado mas que por la integridad de cualquier componente individual. Tener presente que la inconsistencia en las reglas operativas de un sistema pueden conducir una falla a nivel de sistema. Estas fallas también incluyen *fallas de operador*, en las cuales un operador no realiza correctamente su rol en la operación del sistema. Los sistemas que distribuyen objetos o información son propensos a una clase especial de falla de sistema: *fallas de réplicas*. Las fallas de réplicas suceden cuando la información replicada en un sistema se vuelve inconsistente, tanto porque las réplicas que se suponen que proveen resultados idénticos ya no lo hacen o, porque los conjuntos de datos de distintas réplicas no

son consistentes con las especificaciones del sistema. Las fallas de réplicas pueden ser causadas por fallas maliciosas, en las cuales componentes tales como sensores “mienten” al proveer versiones conflictivas de la misma información a otros componentes del sistema. Las fallas maliciosas muchas veces se conocen como fallas bizantinas, con relación al problema de los generales bizantinos que querían tener consenso sobre un ataque cuando uno de los generales era un traidor [LSP82].

Consideremos los sistemas de computación en un automóvil. Supongamos que la computadora para expansión del *airbag* y la computadora de frenos antibloqueo se sabe que funcionan bien y aún así fallan porque una interfiere con la otra cuando ambas están presentes.

Fallas Externas

Las fallas externas aparecen desde fuera de los límites del sistema, el entorno, o el usuario. Las fallas ambientales incluyen fenómenos que afectan directamente la operación del sistema, tales como temperatura, vibración o radiación electromagnética o que afectan las entradas que provistas al sistema. Las fallas del usuario son creadas por el usuario que emplea el sistema. Tener presente que los roles de usuario y operador se consideran por separado; el usuario se considera externo al sistema mientras que el operador se considera como parte del mismo.

Efectos

Las fallas también pueden clasificarse de acuerdo a sus efectos sobre el usuario o servicio del sistema.

Ya que los componentes del sistema de computación interactuan intercambiando datos en un determinado tiempo y/o secuencia, los efectos de las fallas pueden separarse claramente en fallas temporales y fallas de valores. Las fallas temporales ocurren cuando un valor se envía antes o después del momento

especificado. Las fallas de valores ocurren cuando los datos difieren en valor de la especificación.

FALLAS DE VALORES

Los sistemas de computación se comunican proveyéndose valores. Una falla de valor ocurre cuando un cómputo devuelve un valor que no cumple la especificación del sistema. Las fallas de valores se detectan generalmente usando el conocimiento de los valores permitidos para los datos, determinados posiblemente en tiempo de ejecución.

FALLAS TEMPORALES

Una *falla temporal* ocurre cuando un proceso o servicio no se presta o completa dentro del intervalo de tiempo especificado. Una falla temporal no ocurre si no hay especificación explícita o implícita de una meta. Las fallas temporales se pueden detectar observando el momento en el cual acontece una interacción requerida; generalmente no se necesitan conocer los datos involucrados.

Duración

Las *fallas persistentes* permanecen activas durante un significativo período de tiempo. Estas fallas algunas veces se conocen como fallas duras. Las fallas persistentes a menudo son las más fáciles de detectar y diagnosticar, pero pueden ser difíciles de contener y enmascarar a menos que se disponga de hardware redundante. Las fallas persistentes pueden detectarse con facilidad por rutinas de prueba intercaladas con el procesamiento normal. Las fallas transitorias permanecen activas por un período corto de tiempo. Una falla transitoria que se hace activa periódicamente es una falla periódica (algunas veces conocida como falla intermitente). Por su corta duración, las fallas transitorias a menudo se detectan a través de las faltas que resultan de su propagación.

Mecanismos de Tolerancia de Fallas

Los sistemas digitales de computación tienen características especiales que determinan cómo estos sistemas fallan y que mecanismos de tolerancia de fallas

son apropiados. Primero, los sistemas digitales de computación son discretos. A diferencia de sistemas continuos, tales como sistemas de control analógicos, operan en pasos discontinuos. Segundo, los sistemas digitales codifican la información. A diferencia de sistemas continuos, los valores se representan por series de símbolos codificados. Tercero, los sistemas digitales pueden modificar su comportamiento basados en información que ellos procesan.

Ya que los sistemas digitales son sistemas discretos, los resultados pueden probarse o compararse antes de que sean liberados al mundo exterior. Mientras los sistemas analógicos deben aplicar continuamente valores límites o redundantes, un sistema digital puede sustituir un resultado alternativo antes de enviar un valor de salida. Mientras es posible construir computadoras digitales que operen asincrónicamente (sin un reloj principal para secuenciar las operaciones internas), en la práctica todas las computadoras digitales se secuencia a partir de una señal de reloj. Esta dependencia a un reloj hace que una fuente de reloj exacta sea tan importante como una fuente de alimentación, pero también significa que secuencias idénticas de instrucciones llevan esencialmente la misma cantidad de tiempo. Una de los mecanismos de tolerancia de fallas más común, el tiempo de espera, usa esta propiedad para medir la actividad (o falta de actividad) de un programa.

El hecho de que los sistemas digitales codifiquen la información es extremadamente importante. La implicación más importante de la codificación de información es que los sistemas digitales pueden guardar información en forma segura por un largo período de tiempo, una capacidad no disponible en sistemas analógicos, que están sujetos a deriva de valores. Esto significa también que los sistemas digitales pueden guardar copias idénticas de información y esperar que las copias guardadas sean idénticas todavía después de un período de tiempo substancial. Esto hace posible el uso de las técnicas de comparación que se verán más adelante.

La codificación de información en sistemas digitales puede ser redundante, con varios códigos representando el mismo valor. La codificación redundante es la herramienta más poderosa disponible para asegurar que la información en un sistema digital no ha sido cambiada durante el almacenamiento o transmisión. La codificación redundante puede implementarse en varios niveles en un sistema digital. En los niveles inferiores, patrones de código cuidadosamente diseñados anexados a bloques de información digital pueden permitir que hardware de propósito general corrija un número de fallas diferentes de comunicación o almacenamiento, incluyendo cambios a bits únicos o cambios a varios bits adyacentes. La paridad en memorias RAM es un ejemplo común del uso de esta codificación. Ya que un solo bit de información puede tener consecuencias significativas en niveles superiores, un programador puede codificar información sensible, tal como indicadores de modos críticos, con símbolos especiales, imposibles de crearse con error en un único bit.

Administración de la Redundancia

La tolerancia a fallas a veces se llama administración de redundancia. Para nuestros propósitos, redundancia es la provisión de capacidades funcionales que serían innecesarias en un entorno libre de fallas. La redundancia es necesaria, pero no suficiente para la tolerancia a fallas. Por ejemplo, un sistema de computación puede proveer funciones redundantes o salidas tales que al menos un resultado es correcto en presencia de una falla, pero si el usuario debe de algún modo examinar los resultados y elegir el correcto, entonces la tolerancia a fallas sólo está siendo realizada por el usuario.

Sin embargo, si el sistema de computación selecciona correctamente el resultado redundante correcto para el usuario, entonces el sistema de computación no solo es redundante, sino que también es tolerante a fallas. La administración de redundancia reúne los recursos sin fallas para proveer el resultado correcto. La administración de redundancia o tolerancia a fallas involucra una serie de acciones (Tabla 1).

Acción	Características
Detección de Fallas	El proceso de determinar que ha ocurrido una falla.
Diagnóstico de Fallas	El proceso de determinar qué causó la falla, o exactamente que subsistema o componente es defectuoso.
Represión de Fallas	El proceso que previene la propagación de fallas desde su origen en un punto en el sistema a un punto donde puede tener efectos en el servicio al usuario.
Enmascaramiento de Fallas	El proceso de asegurar que sólo valores correctos pasarán a los límites del sistema a pesar de un componente fallado.
Compensación de Fallas	Si una falla ocurre y está confinada a un subsistema, puede ser necesario que el sistema provea una respuesta para compensar la salida del subsistema defectuoso.
Reparación de Fallas	El proceso en el cual las fallas son quitadas del sistema. En sistemas tolerantes a fallas bien diseñados, las fallas se reprimen antes de que se propaguen al extremo de que la provisión del servicio del sistema se vea afectada. Esto deja una porción de sistema inutilizable por fallas residuales. Si ocurren fallas subsecuentes, el sistema puede ser incapaz de hacer frente a la pérdida de recursos, a menos que estos recursos sean recuperados a través de un proceso de recupero el cual asegure que no quedan fallas en los recursos o en el estado del sistema.

Tabla 1 - Acciones para Administrar la Redundancia

La implementación de las acciones descriptas arriba depende de la forma de redundancia empleada tales como redundancia de espacio o de tiempo.

REDUNDANCIA DE ESPACIO

La *redundancia de espacio* provee copias físicas separadas de un recurso, función o ítem de datos. Ya que ha sido relativamente fácil predecir o detectar fallas en unidades de hardware individuales, como procesadores, memorias y vínculos de comunicaciones, la redundancia de espacio es la aproximación usada con más frecuencia asociada con tolerancia de fallas. Es efectiva al tratar con fallas persistentes, tales como averías permanentes de componentes. La redundancia de espacio también es la elección cuando se requiere enmascaramiento de fallas,

ya que los resultados redundantes están disponibles simultáneamente. El interés principal al administrar redundancia de espacio es la eliminación de averías causadas por una falla de una función o recurso que es común a todas las unidades redundantes en espacio.

REDUNDANCIA DE TIEMPO

Como se mencionó antes, los sistemas digitales tienen dos ventajas únicas respectos de otros tipos de sistemas, incluyendo sistemas eléctricos analógicos. Primero, pueden cambiar funciones al almacenar información y programas para manipular la información. Esto significa que si las fallas esperadas son transitorias, una función puede ejecutarse nuevamente con una copia almacenada de los datos de entrada lo suficientemente separada de la primera ejecución de la función para que una falla transitoria no afecte a ambas. Segundo, ya que los sistemas digitales codifican la información como símbolos, pueden incluir redundancia en el esquema de codificación de dichos símbolos.

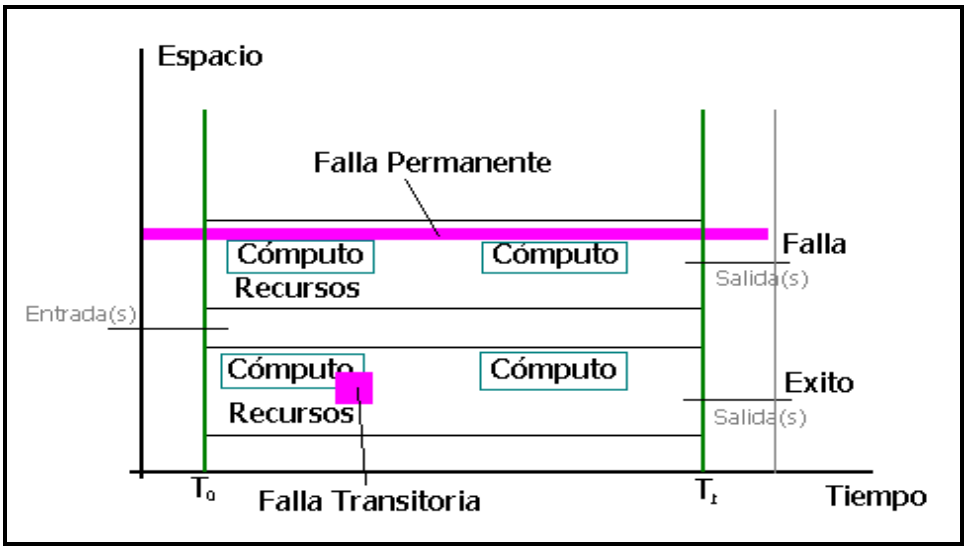

Figura 4 - Redundancias Espacial y Temporal

Esto significa que la información movida en el tiempo puede ser controlada por cambios no deseados, y en muchos casos, la información puede corregirse a su valor original. La Figura 4 ilustra la relación entre redundancia de tiempo y de espacio.

RELOJES

Muchos mecanismos de tolerancia a fallas, que emplean tanto redundancia de tiempo como redundancia de espacio, se basan en una fuente exacta de tiempo. Probablemente ninguna característica de hardware tiene un efecto mayor sobre los mecanismos de tolerancia a fallas como un reloj. Una decisión inicial en el desarrollo de un sistema de tolerancia a fallas debería ser la decisión de disponer de un servicio de tiempo confiable a través de todo el sistema. Tal servicio puede ser usado como una base para protocolos de detección y reparación de fallas. Si el servicio de tiempo no es tolerante a fallas, se deben agregar entonces temporizadores adicionales o se deben implementar protocolos asincrónicos complejos para confiar en el progreso de ciertos cálculos para proveer una estimación del tiempo. Los diseñadores de sistemas multiprocesadores deben decidir si proveen un servicio de reloj global tolerante a fallas que mantenga una fuente de tiempo consistente a través de todo el sistema, o resolver los conflictos de tiempo sobre una base ad hoc [Lam85].

CODIFICACIÓN

La codificación es el arma principal en el arsenal de tolerancia a fallas. Las decisiones de codificación a bajo nivel son tomadas por los diseñadores de procesadores y memorias cuando ellos eligen los mecanismos para detección y corrección de errores para las memorias y buses de datos. Los protocolos de comunicaciones proveen una variedad de opciones de detección y corrección, incluyendo la codificación de grandes bloques de datos para soportar múltiples fallas contiguas y la provisión de reintentos múltiples en caso de que las capacidades de corrección de errores no puedan hacer frente a las fallas. Estas capacidades deberían complementarse con técnicas de codificación de alto nivel que registren valores críticos del sistema usando patrones únicos que sean improbables de crearse por azar.

Técnicas de Comparación

DETECCIÓN DE FALLAS

La comparación es una alternativa a las pruebas de aceptación para detectar fallas. Si la fuente principal de fallas es el procesador (hardware), entonces se usan múltiples procesadores para ejecutar el mismo programa. A medida que los resultados se calculan, se comparan a través de los procesadores. Una discrepancia indica la presencia de una falla. Esta comparación puede estar basada en pares o involucrar tres o más procesadores simultáneamente. En el último caso, el mecanismo usado generalmente se conoce como *votación*. Si las fallas de diseño de software son una preocupación principal entonces la comparación se hace entre los resultados de múltiples versiones del software en cuestión, un mecanismo conocido como programación de N-versiones (*N-version programming*) [Che78].

DIAGNÓSTICO DE FALLAS

El diagnóstico de fallas con comparación depende de si es usada comparación basada en pares o por votación. La tabla a continuación describe los efectos de dichos métodos:

Método de Comparación	**Efecto**
Basada en pares	Cuando ocurre una discrepancia en un par, es imposible decir cual de los procesadores ha fallado. El par entero debe declararse defectuoso.
Votación	Cuando tres o más procesadores ejecutan el mismo programa, el procesador cuyos valores no concuerdan con los otros se diagnostica prematuramente como defectuoso.

REPRESIÓN DE FALLAS

Cuando se usa comparación basada en pares, la represión se alcanza parando toda actividad en el par que no concuerda. Cualquier otro par en operación puede continuar ejecutando la aplicación, sin disturbios. Ellos detectan la avería del par que no concuerda a través de tiempos de espera (*time-outs*). Cuando se usa

votación, la represión se logra ignorando el procesador averiado y reconfigurándolo fuera del sistema.

ENMASCARAMIENTO DE FALLAS

En un sistema basado en comparación, el enmascaramiento de fallas puede alcanzarse por dos caminos. Cuando se usa votación el votante permite sólo que pase el valor correcto. Si se usan votantes de hardware, esto sucede generalmente lo suficientemente rápido para cumplir cualquier meta de respuesta. Si la votación se hace por votantes de software que deben alcanzar consenso, puede no estar disponible el tiempo adecuado.
La comparación basada en pares requiere que la existencia de múltiples pares de procesadores para enmascarar averías. En este caso el par de procesadores averiado se detiene, y los valores son obtenidos de pares funcionales buenos.

REPARACIÓN DE FALLAS

En un sistema basado en comparación con un único par de procesadores, no hay recuperación de una avería. Con múltiples pares de pares, la recuperación consiste en el uso de valores del par “bueno”. Algunos sistemas proveen mecanismos para reiniciar el par fuera de comparación con datos del par "bueno”. Si ese par fuera de comparación produce resultados que comparan durante un período de tiempo adecuado, puede ser configurado nuevamente en el sistema.
Cuando se usa votación, la recuperación de un procesador averiado se lleva a cabo utilizando los valores “buenos” de los otros procesadores. A un procesador desplazado por votación se le permitirá continuar la ejecución y puede ser configurado nuevamente en el sistema si coincide exitosamente en un número especificado de votaciones subsecuentes.

Diversidad

La única aproximación de tolerancia a fallas para combatir los errores de diseño comunes es la diversidad de diseño – la implementación de más de una variante de la función a realizar [Avi85]. Para aplicaciones basadas en computadoras, generalmente se acepta que es más fácil variar un diseño en niveles más altos de

abstracción (p. e. variando el algoritmo o los principios físicos usados para obtener un resultado) que variar los detalles de implementación de un diseño (p. e. usando distintos lenguajes de programación o técnicas de codificación de bajo nivel). Ya que diversos diseños deben implementar una especificación común, la posibilidad de dependencias siempre aparece en el proceso de refinar la especificación para reflejar las dificultades no cubiertas en el proceso de implementación. Diseños verdaderamente diversos eliminarían las dependencias en equipos comunes de diseño, filosofías de diseño, herramientas de software y lenguajes, y aún filosofías de prueba. Muchas aproximaciones intentan lograr la independencia necesaria a través del azar, al crear equipos de diseño separados que permanecen físicamente separados a través de los procesos de diseño, implementación y prueba. Recientemente, algunos proyectos han intentado crear diversidad al forzar diferentes reglas de diseño para varios equipos.

Resumen

Un sistema de tiempo real está condicionado por el tiempo de sus respuestas. Consta de tareas que realizan procesos de monitoreo (mediante sensores), control (rutinas de control) y actuación (a través de actuadores). La planificación de estas tareas es esencial para garantizar el cumplimiento de sus metas (restricciones de tiempo). Existen métodos formales de especificación y diseño, ya que su complejidad es cada vez mayor. El campo de acción de este tipo de sistemas es creciente y la tolerancia a fallas debiera ser una cualidad intrínseca en ellos.

La tolerancia a fallas es una cualidad que un sistema debe tener para ser confiable, y los sistemas de tiempo real, por su naturaleza y ámbito de aplicación, deben ser confiables. Es un área que con el tiempo a alcanzado mucho desarrollo y una creciente aceptación, con gran disponibilidad de estrategias, técnicas y conceptos bien establecidos. Tiene un uso particularmente aceptado en sistemas distribuidos. Veremos cómo muchas de esas técnicas se están integrando en sistemas de tiempo real. Información adicional sobre conceptos y terminología de tolerancia a fallas puede encontrarse en [And81], [And82] y [Lap90].

3. Tolerancia a Fallas en Sistemas de Tiempo Real
Estado del Arte

"La palabra 'imposible' no está en mi vocabulario"

Napoleón

Introducción

En las secciones precedentes se introdujeron conceptos necesarios para comprender el alcance de este trabajo, ya sea tanto en el rubro de sistemas de tiempo real como de tolerancia a fallas, pero por separado. Llegado este punto, es momento de presentar los hallazgos realizados por diversos grupos de investigación sobre tolerancia a fallas aplicable a distintas áreas dentro de los sistemas de tiempo real.

Sin intención de que tuviese alcance enciclopédico, la búsqueda de información se limitó entonces a tres sectores bien diferenciados de los sistemas de tiempo real, los cuales serán presentados en el siguiente orden: la especificación y diseño, la planificación de tareas y el uso de sensores y actuadores. En el primer caso se presentan variados ejemplos de aplicación de técnicas formales para la especificación de sistemas de tiempo real que permiten incluir "desde el inicio" características de tolerancia a fallas. La idea tras estos trabajos es generar un sistema de tiempo real tolerante a fallas, previendo los problemas desde el instante mismo de la concepción de los requisitos a cumplir por dicho sistema, con la intención de llegar a evitar (o al menos reducir) los inconvenientes por volver a diseñarlo para tolerar fallas.

Por su parte, la planificación de tareas es en sí misma un tema de investigación, y que se revela con gran interés por parte de la comunidad académica. Por lo tanto, sólo se presentarán varios métodos de planificación mostrando sus variantes tolerantes a fallas; junto con las ventajas y desventajas de esos métodos respecto

de los que les dieron origen, y comentarios sobre las posibles áreas de trabajo en ese sentido.

Luego será el turno del uso de sensores y actuadores. Este sector se muestra también con gran actividad de investigación, al ser las características de tolerancia a fallas muy importantes, sobre todo en sistemas que incluyan gran cantidad de sensores y actuadores. Muchos de los trabajos revisados tienen gran relación con la robótica y además se hace notorio el uso de la inteligencia artificial (por ejemplo, mediante el uso de redes neuronales o comportamientos para enmascarar o reaccionar ante fallas).

Especificación y Diseño

Verificación de Tolerancia a Fallas y Tiempo Real

El uso de lógica temporal en la especificación de sistemas de tiempo real ya se ha estado usando desde hace varios años. Estas técnicas permiten expresar las restricciones de tiempo para las acciones a llevar a cabo por un sistema en particular y facilitan además la verificación formal ya que el uso de grafos temporizados (una composición de autómatas y variables reales a modo de relojes) brinda la capacidad de la descripción de propiedades temporales junto a un método de verificación apropiado para la modelización de ejemplos de la vida real. La existencia de un método de transformaciones para especificar y verificar programas del tiempo real tolerantes a fallas mediante una lógica con expresividad suficiente para modelar los cambios de estado debido a fallas en el sistema hace factible la determinación a priori de sus efectos en las propiedades funcionales y de tiempo real de esas acciones.

La estructura de transformaciones presentada en [Zhi96] asume que las fallas físicas (que van desde procesadores que fallen, memoria corrupta, canales con problemas de comunicaciones, sensores o actuadores atascados, etc.) de un sistema se modelan como siendo causadas por una serie de operaciones “con fallas” que realizan una transformación de estado del mismo modo que lo hacen

las operaciones normales del sistema. Con el lenguaje de especificación y el modelo computacional propuestos en el trabajo en cuestión, se arman los llamados modelos de acciones del sistema. Estos incluyen una *condición inicial* y una serie de *operaciones atómicas* sobre un conjunto finito de *variables de estado*. Basados en TLA (*T*emporal *L*ogic *A*ction) [Lam90], se puede razonar formalmente sobre las propiedades de las acciones de un sistema, ya que se demuestra existe una relación directa entre las acciones del sistema y las fórmulas en TLA. Estos componentes son la base para introducir la tolerancia a fallas al incorporar los conceptos de falla física, error, operación con falla y entorno con fallas. Con la capacidad entonces de describir y manejar formalmente programas tolerantes a fallas, se extiende el modelo para permitir el tratamiento de metas que imponen restricciones temporales a las acciones, las cuales deben ejecutarse ni muy temprano ni muy tarde (esto involucra la aparición de condiciones de tiempo límite inferior y tiempo límite superior). Para refinar el modelo propuesto y lograr la tolerancia a fallas en un sistema de tiempo real, se necesita asumir que la ocurrencia de fallas, especialmente cuando se necesitan cumplir las metas. En general, estas suposiciones son una restricción a la frecuencia en que ocurrirán las fallas y la imposición de que siempre que suceda la falla 1 no puede ocurrir la falla 2 dentro de x unidades de tiempo. Este trabajo demuestra como mediante transformaciones, el refinamiento paso a paso puede usarse para desarrollar sistemas tolerantes a fallas, con o sin restricciones de tiempo.

Evaluación de la Latencia de Tolerancia a Fallas desde la Perspectiva de Aplicaciones de Tiempo Real

[Kim94] define la *latencia* de tolerancia a fallas (FTL - *F*ault *T*olerance *L*atency) como el tiempo requerido por todos los pasos secuenciales llevados a cabo para recuperar de un error. Esta definición puede volverse sumamente importante en sistemas de tiempo real, donde la latencia de cualquier política de manejo de errores (tales como detección, enmascaramiento y recuperación de errores, por ejemplo) no debe ser superior a la latencia requerida por la aplicación (ARL - *A*pplication *R*equired *L*atency), que es un tiempo que depende del proceso

controlado bajo estudio y su entorno. Al evaluar la FTL considerando distintos mecanismos de tolerancia a fallas (Figura 5), se puede utilizar dicho FTL para comprobar si una determinada estrategia de tolerancia a fallas elegida para una aplicación puede cumplir las metas (debe ser FTL ≤ ARL).

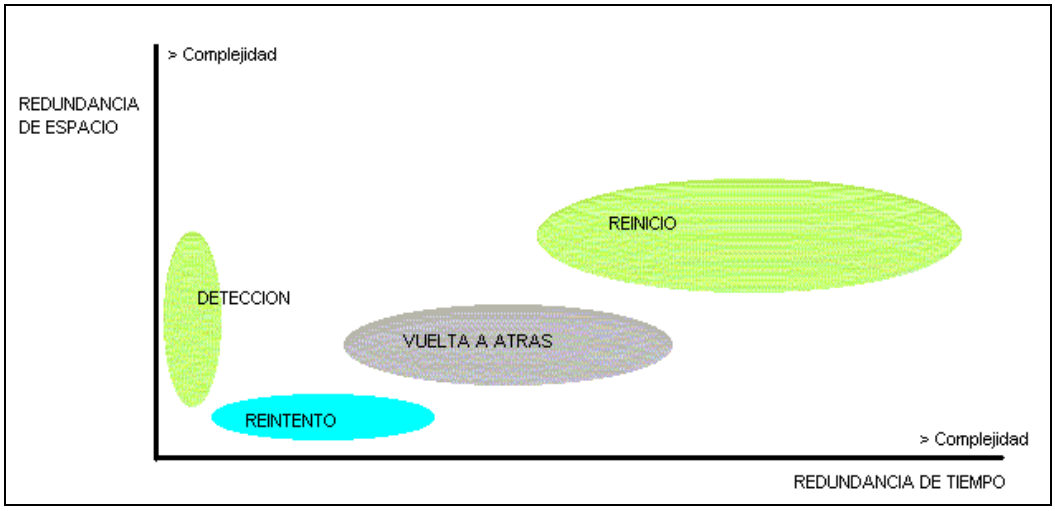

Figura 5 - Costo/Beneficio entre la redundancia espacial y temporal para diversos esquemas de tolerancia a fallas

Un concepto así se aplica en las etapas iniciales del diseño de una aplicación de tiempo real a la cual se quiere proveer de capacidades de tolerancia a fallas. Al utilizar el valor ARL del proceso controlado y el valor FTL del proceso de control, se está en posición de seleccionar o diseñar una estrategia de tolerancia a fallas apropiada, que permite además realizar un balance entre la redundancia espacial y temporal mientras se satisfacen las restricciones de tiempo. Según los autores, este método aún necesita maneras de evaluar o modelar los tiempos que se consumen en las etapas individuales de la recuperación mientras se consideran la arquitectura del sistema, el tipo de tarea y la implementación de cada etapa. Este método depende de la cobertura de fallas, que se asume es una constante determinada en la etapa de diagnóstico; si bien dicho valor no es tan simple de calcular. La cobertura de fallas está afectada por los tipos de tareas y el régimen de ocurrencia de fallas, por lo cual es importante estudiar esos factores y sus efectos sobre el valor final de FTL para la aplicación a desarrollar.

Especificación Formal y Verificación de un Modelo para Enmascaramiento de Fallas y Recuperación para Sistemas Digitales de Control de Vuelo

Las modernas aeronaves comerciales y militares dependen de sistemas de control de vuelo digitales[2] (DFCS - *D*igital *F*light *C*ontrol *S*ystems); sistemas que interpretan las entradas de control del piloto y envían los comandos apropiados para las superficies de control y los motores. Dependiendo del tipo de avión, dichos sistemas pueden tener o no sistemas redundantes compuestos por computadoras analógicas o sistemas convencionales mecánicos e hidráulicos[3]. Una de las ventajas inherentes a los DFCS es la economía y rendimiento, ya que la eliminación de pesados vínculos de control mecánicos e hidráulicos reduce el peso de la aeronave y aumenta la capacidad de carga y/o mejora la eficiencia del combustible. Un control óptimo del motor junto a la mejor actitud en todo momento del vuelo también reduce en forma significativa el consumo de combustible.

Ya existen aviones diseñados de manera tal que son intrínsecamente inestables, y que sólo pueden volar mediante el control por computadora. Otra ventaja aportada por estos sistemas es la seguridad, ya que evitan que el piloto coloque al avión en maniobras imposibles de recuperar; o lo ayudan a salir de una situación (pérdida de una superficie de control o falla en un motor) en la cual el piloto sería incapaz de aprender como controlarlo en el poco tiempo disponible.

[2] El popular término FBW (fly-by-wire) cubre tanto DFCSs como los anteriores (y similares) sistemas que empleaban computadoras analógicas (por ejemplo, el Concorde). Los sistemas FBL (fly-by-light) son simplemente sistemas FBW en los cuales los cables de cobre han sido reemplazados por fibra óptica.

[3] El Concorde (1969) tenía sistemas redundantes mecánicos en todos los tres ejes primarios; el Boeing 777 (1996) no tiene sistemas mecánicos redundantes en ningún eje.

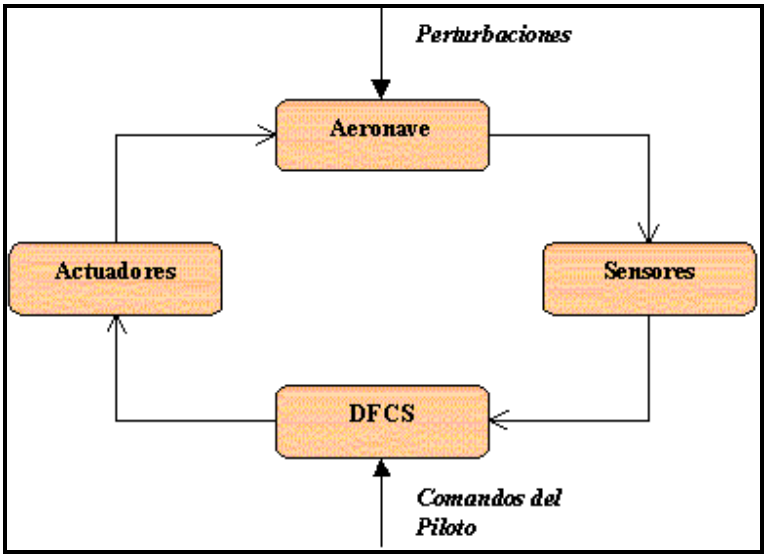

Figura 6 - Esquema básico de un DFCS

Como puede apreciarse en la Figura 6, un DFCS no ni más ni menos que un sistema de tiempo real. Es por eso que los hallazgos sobre tolerancia a fallas del trabajo presentado en [Rus91] tienen aplicación en esta tesis. Dada su utilización en aeronaves, queda asumida la intrínseca capacidad de tolerancia a fallas requerida a estos sistemas. El propósito fue desarrollar la especificación y verificación formal de un modelo apropiado para las computadoras replicadas en los DFCS. Obviamente, estos formalismos pueden usarse directamente en el diseño de otros tipos de sistemas reales. El resultado es que, mediante una serie de teoremas (que han sido verificados mecánicamente por el método EHDM) las fallas entre los componentes computacionales de un DFCS pueden enmascararse, es decir, los comandos enviados a los actuadores serán los mismos que aquellos que enviaría una única computadora que no sufra fallas.

Durante el documento se enuncian los antecedentes de los DFCS y se presentan varios ejemplos de aviones experimentales y algunas situaciones encontradas durante las pruebas de vuelo de dichas aeronaves. Se presentan también modelos formales para los DFCS y las características propias de un modelo para enmascarar fallas, todo desde un punto de vista formal. Este modelo se especifica y verifica y es comparado con el desarrollo conocido como LaRC [DiV90]; su logro es mostrar que ciertos principios de diseño (distribución bizantina tolerante a fallas

para muestras de sensores, ejecución sincronizada, votación por mayoría de las salidas para todos los actuadores y votación por mayoría de los datos del estado interno), proveen un comportamiento predecible que enmascaran las fallas y permiten recuperación. Estos principios de diseño se codifican con los axiomas y definiciones de ese modelo.

Planificación

Garantía de Tolerancia a Fallas mediante Planificación

[Gho96] exhibe una tesis que tuvo como propósito comprobar que mejorar las capacidades de tolerancia a fallas en sistemas de tiempo real puede lograrse sin cambios importantes al hardware (sin equipo especializado ni excesivamente redundante) o al software (reescribiendo las aplicaciones del usuario). Para ello se presentan distintos modelos de esquemas de planificación, que aseguran la tolerancia a fallas.

Aún cuando varios tipos de fallas pueden tolerarse al agregar redundancia al sistema, con sólo agregar redundancia no es suficiente. Dichos recursos adicionales tienen que administrarse de forma tal que todas las restricciones temporales se cumplan, y se garantice que las fallas sean toleradas. Para mantener esas garantías (tolerancia a fallas y metas), se necesitan algoritmos de planificación especializados.

El objetivo principal perseguido fue garantizar la recuperación de tareas de tiempo real que no generan los resultados correctos debido a fallas en el sistema. Eso se logra planificando una tarea asegurando que la disponibilidad de suficiente tiempo para permitir la re-ejecución de la tarea defectuosa dentro de su meta y sin comprometer las restricciones temporales de cualquier otra tarea. Además, se estudió el efecto de agregar redundancia al sistema, ya que si se aumenta la redundancia, más fallas pueden tolerarse, pero a costa de disminuir el porcentaje de recursos que se usan para operaciones concretas. Se midió la relación costo-beneficio para distintos modelos de tareas al variar la cantidad de redundancia en el sistema. Otro aspecto importante de este trabajo es resaltar los objetivos secundarios, a saber:

- Mejorar la capacidad de tolerancia a fallas de sistemas de tiempo real usando redundancia temporal y describir los esquemas de recuperación para tareas con fallas.

- Proporcionar las garantías requeridas para la ejecución tolerante a fallas en sistemas de tiempo real.
- Estudiar la relación costo-beneficio entre la capacidad de tolerancia a fallas y la utilización del sistema.
- Usar técnicas para mejorar la utilización del sistema.
- Usar técnicas para mejorar la eficiencia de los algoritmos de planificación de tiempo real.
- Medir la elasticidad (*resiliency*) de los sistemas de tiempo real, expresada como la habilidad de un sistema de tolerar una segunda falla luego de recuperarse de una primera.
- Emplear varias técnicas de análisis para estudiar el rendimiento de los esquemas de planificación.

Al usar redundancia de tiempo en los algoritmos de planificación de tareas para tiempo real, se garantiza que todas las tareas puedan satisfacer sus restricciones temporales al manejar el tiempo de procesamiento de forma adecuada para tratar las demoras en la recuperación de fallas. Los modelos propuestos (uni- o multiprocesadores) están acompañados de pruebas que proveen garantías con relación al número y frecuencia de las fallas que pueden tolerarse. Esas pruebas se basan en usar límites de utilización (en sistemas con remoción de tareas) o un conjunto de condiciones (en sistemas sin remoción de tareas). Se necesitó una medida para ponderar la eficiencia de esas técnicas de planificación y se optó por emplear el porcentaje de tareas entrantes que pueden planificarse.

Los mecanismos de detección de fallas usados en este trabajo se desarrollaron para sistemas que no son de tiempo real, con lo no consideran el tiempo empleado en detectar las fallas. Sin embargo, para su utilización en sistemas de tiempo real, es importante limitar el tiempo usado en la detección de fallas, y ese tema es tópico de trabajo posterior. Otra área para abordar en futuros trabajo es la extensión de los esquemas de planificación para sistemas de tiempo real distribuidos. El gran desarrollo de las redes de computadoras hace de éste un

campo con gran potencial para obtener logros concretos. En este caso, el planificador de un sistema distribuido no debería tener una visión global del sistema, sino considerar la planificación de tareas, y sus copias de resguardo, trabajando con un conjunto limitado de vecinos.

Tasa Monotónica Tolerante a Fallas

La política de planificación desarrollada por Liu y Layland [Liu73] conocida como Tasa Monotónica (RMS - *R*ate *M*onotonic *S*cheduling) ha sido usada en forma extensa en sistemas de tiempo real. Sin embargo, en su versión original, este algoritmo no provee mecanismos para administrar redundancia de tiempo, de modo que las tareas de tiempo real puedan cumplir sus metas aún en presencia de fallas. Eso es lo que motivó el trabajo expuesto en [Gho97], cuyo objetivo fue extender el esquema RMS para proveer tolerancia a fallas transitorias (únicas o múltiples). Se demuestra que tal extensión es factible, y a ese nuevo esquema se lo denomina FT-RMS.

La aproximación general para la tolerancia a fallas usada por los autores es estar seguros de que hay un período de poca actividad (slack) suficiente en la planificación de modo que si ocurre una falla mientras cualquiera de las tareas se está ejecutando, esa tarea pueda ser ejecutada nuevamente. Las tareas se ejecutan siguiendo el esquema RMS usual si no ocurren fallas (y el período de poca actividad no se usa). Por lo pronto, cuando sucede una falla en una tarea, se emplea un esquema de recuperación para volver a ejecutar la tarea. La Figura 7 presenta un ejemplo de tareas planificadas y tiempo de reserva.

Aprovechando la característica de utilización del procesador menor al 100% del algoritmo RMS, naturalmente habrá un período de poca actividad. Pero para incorporar la tolerancia será necesario poder calcular ese período en forma determinística. Inicialmente valen todas las definiciones y características impuestas por el método RMS, pero este trabajo agrega el concepto de tarea de resguardo (*backup*) junto con las condiciones necesarias para asegurar la

recuperación en un sistema periódico con remoción mediante una serie de lemas y pruebas, y cuyo punto más importante es quizás el estudio del efecto del período de poca actividad en el factor de utilización, puesto que es natural observar que dicho factor decrecerá cuando una fracción de la capacidad de procesamiento se reserve para le re-ejecución. Sin embargo, se establecen una serie de pasos para calcular un límite de utilización (tolerante a fallas) el cual es función del número de tareas a planificar, el tiempo de recupero a utilizar y el límite superior de utilización.

FT-RMS garantiza que se toleran fallas (únicas o múltiples) para cada instancia de cada tarea planificada, si se reserva suficiente períodos de poca actividad. Y la cantidad de fallas toleradas es función de la cantidad de tiempo reservada. Se describe un esquema de recuperación sencillo en el cual la tarea defectuosa se vuelve a ejecutar con su misma prioridad. Como se detalla dicho esquema y además se aportan las condiciones y pruebas necesarias, cualquier sistema que use RMS puede usar este esquema de tolerancia a fallas. El método de cálculo de los límites de utilización para un conjunto de tareas, puede usarse para determinar si las tareas son planificables con tolerancia a fallas.

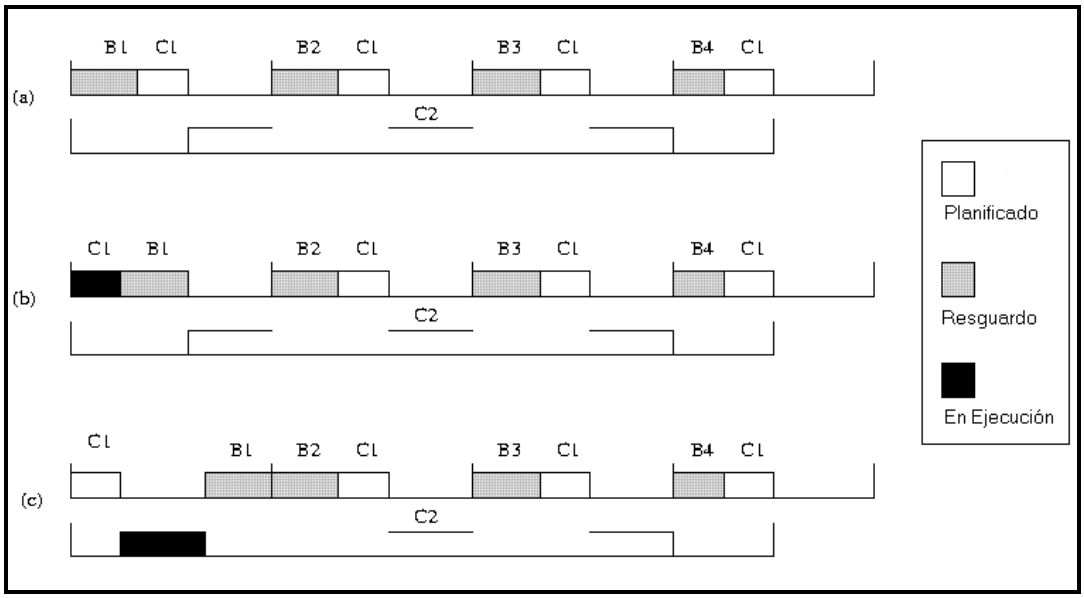

Figura 7 - Planificación RMS con tolerancia a fallas

Por su parte el esquema de recuperación también puede servir para:

- tolerar fallas temporales en una tarea, las cuales suceden cuando una tarea excede su tiempo máximo asignado
- implementar el paradigma de cómputo impreciso

En ambos casos, el período de poca actividad reservado puede usarse para completar la ejecución de la tarea, y de esta forma tolerar una falla temporal (de duración igual al tiempo de recupero, o al menos dos veces el tiempo de ejecución).

Integrando Planificación y Tolerancia a Fallas en Sistemas de Tiempo Real

En [Sta92b] se repasan varios otros trabajos relacionados con el tema de conjugar la complicada tarea de planificación de tiempo real con la también difícil empresa de tolerancia a fallas, resumiendo en cada uno las principales características o logros. Para describir el impacto de la integración de planificación y tolerancia a fallas, se presenta un modelo de sistema y se describe una tarea (con las características usuales como ser si es periódica o no, removible o no, la importancia de la tarea, el peor tiempo de cálculo, etc.) junto a una semántica de tolerancia a fallas que incluyen si la tarea puede ser replicada en forma pasiva o activa con votación, si puede requerir cierto grado de redundancia de tiempo o espacio.

Dicho modelo se compara además con otro modelo, llamado de cálculo impreciso, cuya característica principal es reducir las fallas temporales y las metas no cumplidas al reducir la exactitud de los resultados provistos de manera natural. La tarea contiene una parte mandatoria que debe cumplirse para que se cumpla su meta, y el algoritmo de planificación basado en el cálculo impreciso debe garantizar que se cumplen todas las partes mandatorias de las tareas. Con ello se provee un rendimiento mínimo garantizado aún con sobrecargas, y además el algoritmo intenta ejecutar lo más posible las partes opcionales de las tareas para minimizar el error. Este modelo es factible de extensiones al poder usar diferentes métricas en el planificador.

Por último, se dejan planteadas una serie de cuestiones propias de la integración de la planificación con la tolerancia a fallas, a saber:

- Qué nueva teoría de planificación se necesita para soportar tal integración.
- Cómo puede la planificación dinámica contribuir en la relación costo-beneficio entre redundancia de tiempo y espacio.
- Puede un solo algoritmo sofisticado manejar efectivamente conjuntos de tareas complejas, o tales tareas deben ser partidas en clases de equivalencia, con algoritmos ajustados para cada clase; cómo interactuarían tales conjuntos de algoritmos.
- Qué tipo de predicción, incluyendo garantía de tolerancia a fallas, si es posible para cálculos de tiempo real distribuidos, y si puede ser desarrollado tal algoritmo.
- Cómo pueden combinarse la prioridad de la tarea, tiempo de cálculo, dureza de la meta y requisitos de tolerancia a fallas para maximizar el valor del sistema; cuáles son los roles de los algoritmos de planificación en este análisis.
- Cuál es el impacto de las políticas de reserva fuera de línea y tolerancia a fallas en la planificación dinámica en línea.
- Muchas técnicas de tolerancia a fallas (diagnóstico, recuperación, elección de un nuevo líder, mantenimiento de backups pasivos, migración de tareas, etc.) se desarrollan sin tener en cuenta cuándo planificar estas tareas; ésto tiene implicancias serias respecto de la predicción y cumplimiento de las metas, se necesita entender el impacto de cada una de estas técnicas y la planificación de las mismas en el rendimiento general del sistema.
- Cómo realizar planificación y tolerancia a fallas efectivas en costo.
- Cómo pueden adaptarse las políticas de tolerancia a fallas ya que los sistemas reaccionan en el corto y largo plazo a condiciones cambiantes ya sea en el entorno, los objetivos, requisitos y condiciones de los sistemas.

Estas afirmaciones y preguntas son una buena base de partida para aquellos que deseen explorar las posibilidades de la planificación de tiempo real con tolerancia a fallas.

Sensores y Actuadores

Tolerancia a Fallas en HANNIBAL

Hannibal es un robot autónomo con muchos sensores y actuadores, usado para estudiar el control de un robot complejo, la locomoción en un hexópodo y la tolerancia a fallas de sensores y actuadores, y cuyo desarrollo y conclusiones se presentan en [Fer93]. Pensado como un explorador planetario, este robot debía tener capacidades de control en tiempo real (al recorrer un terreno desconocido en otro planeta, no podía esperar las órdenes de un centro de control terrestre, las cuales llegan varios minutos después de ser emitidas) y además tolerar fallas en hardware y software y continuar con sus tareas (la imposibilidad de reparar los componentes averiados). Muchos de sus conceptos fueron adoptados (y probados realmente) en la misión Pathfinder que exploró Marte en Julio de 1997, a través de la sonda Sojourner.

En particular, es de interés para nuestro trabajo estudiar la aproximación sobre tolerancia a fallas encarada en dicho trabajo. Como base principal, se usó una red de tolerancia a fallas, cuyos objetivos son:

- *Rápido tiempo de respuesta a averías*: al operar en un entorno peligroso, se deben detectar y remediar las averías rápidamente o se puede arriesgar la seguridad del sistema. Esto impone que las averías deban detectarse y compensarse antes de que el rendimiento del sistema se degrade a un nivel inaceptable.
- *Degradación natural del rendimiento*: el rendimiento del sistema se debe degradar naturalmente a medida que se acumulan las averías. Ello implica mantener el más alto nivel de rendimiento posible dado el estado funcional del hardware.
- *Acceso a todos los sensores confiables*: Muchos sensores y actuadores mejoran el rendimiento del sistema siempre que sean funcionales. Por esto, los componentes reparados deberían poder incorporarse nuevamente al uso.

- *Cobertura de fallas*: El sistema puede sufrir una variedad de averías. Pueden ser permanentes o transitorias. Algunas tienen efecto local, mientras que otras tienen alcance global sobre el rendimiento. El sistema debe poder recuperarse de distintas combinaciones de fallas. Las averías pueden ocurrir individualmente, en forma concurrente con otras fallas, o acumularse en el tiempo.

CONFINAMIENTO DE ERRORES

Las fallas de sensores y actuadores afectan varios niveles de la jerarquía del sistema. El nivel de hardware de sensores y actuadores es el nivel más bajo, el control de bajo nivel es el nivel intermedio y el control de comportamiento es el nivel más alto. Se ve claramente que fallas de sensores o actuadores afectan el nivel de hardware del sistema. Las fallas de sensores afectan el control de bajo nivel porque los agentes de sensores virtuales usan información de los sensores reales para calcular sus resultados. Como consecuencia, las fallas de sensores pueden causar resultados incorrectos de los sensores virtuales. Si este es realmente el caso, entonces el control de alto nivel también es afectado por fallas de sensores. Los agentes de sensores virtuales son responsables de activar el comportamiento correcto en el momento apropiado. Sin embargo, si los resultados de los agentes virtuales son incorrectos, entonces se activarán los comportamientos equivocados.

Es importante detectar y confinar los errores al nivel más bajo posible en el que ocurren. Si un error no se confina al nivel en el que se origina, entonces los niveles superiores deben detectar y compensar la falla. Cuando un error se propaga a niveles superiores en la jerarquía, afecta más partes del estado del sistema. Tiempos de respuesta mayores para la corrección de errores significa que las manifestaciones de los errores se harán más diversas. Por tanto, detectar y confinar errores al más bajo nivel posible de la jerarquía del sistema maximiza la efectividad de los procedimientos de recuperación y minimiza el impacto del error en el rendimiento del sistema.

NIVELES DE TOLERANCIA A FALLAS

Dado que las fallas de hardware afectan varios niveles del sistema, las técnicas de tolerancia a fallas pueden implementarse en cada nivel. Se presentan las estrategias posibles para cada nivel, y se describen los méritos y desventajas de cada método.

REPLICACIÓN DE HARDWARE

Ya que los sensores y actuadores son propensos a fallas, se desea un sensado y actuación confiable. Varios sistemas usan la replicación de hardware para mejorar la confiabilidad del hardware y además para lograr robustez. En el caso de Hannibal, esto correspondería a replicar sensores y actuadores. Para un ejemplo de sensado, múltiples potenciómetros podrían usarse para medir un ángulo de unión particular (Figura 8). Un árbitro recolecta los valores del ángulo de cada potenciómetro y usa el valor más común para ajustar el valor de ángulo aceptado. La aplicación usa este valor como el ángulo de unión.

La falla de un sensor de ángulo de unión se detecta si su valor no concuerda con los valores de los otros sensores. Sin embargo, la falla se enmascara porque el valor más común se acepta como el valor real. Asumiendo que la mayoría de los potenciómetros trabajan correctamente, la recuperación de la falla es inmediata porque el software de aplicación usa el valor correcto del ángulo de la unión. En consecuencia, las fallas de hardware se detectaron y confinaron al nivel de hardware de la jerarquía del sistema. Idealmente, se podría detectar, confinar y corregir errores de hardware en el nivel de hardware. Haciendo esto, el software de aplicación no necesita ser alertado de estas fallas. Sin embargo esta aproximación tiene sus inconvenientes. Primero, la replicación de sensores y actuadores es cara en términos de dinero. Segundo, asume que la mayoría de los componentes replicados está funcionando correctamente. Teóricamente, Hannibal debería funcionar confiablemente con una minoría de sensores funcionales. Tercero, las restricciones de tamaño y peso de Hannibal restringen cuántos sensores y actuadores pueden montarse en el robot. En consecuencia, no es

práctico usar este esquema en Hannibal para mejorar las capacidades de sensado y actuación. Sin embargo, Hannibal fue diseñado con múltiples sensores y actuadores para poseer capacidades complementarias.

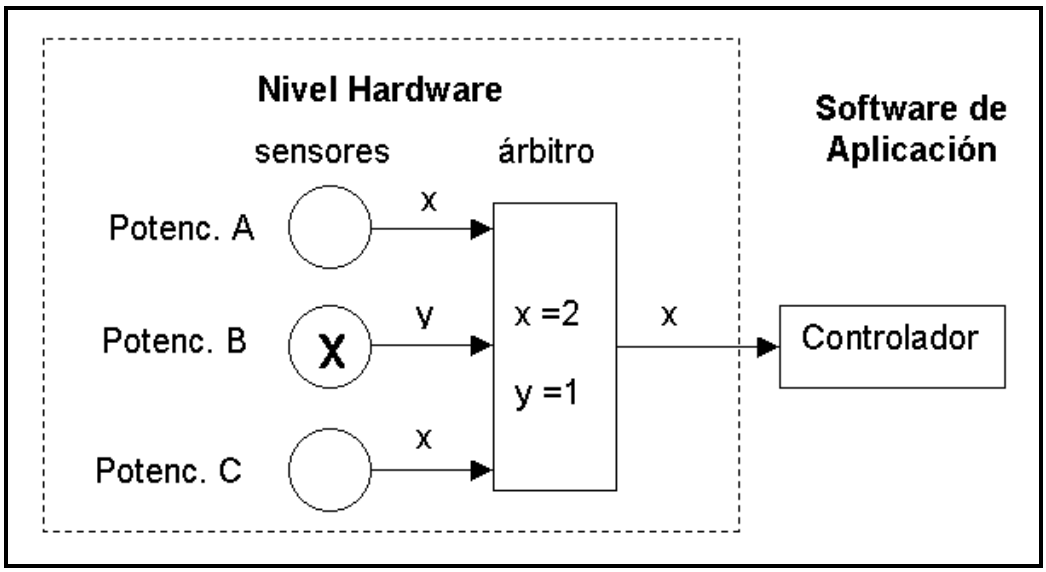

Figura 8 - Replicación de Componentes a Nivel de Hardware

Por ejemplo, Hannibal puede usar varios distintos sensores para detectar el contacto con el terreno, y puede perder la función mecánica de una pata y todavía seguir caminando. El sistema de control de Hannibal debe ser inteligente en la forma que usa los sensores y actuadores existentes para compensar las fallas de sensores y actuadores.

COMPORTAMIENTOS DE CONTROL REDUNDANTES

El método de estrategia redundante implementa tolerancia a fallas en niveles altos de control y se dedica a fallas de alto nivel. Por ejemplo, una falla de alto nivel ocurre cuando el robot encuentra una situación que no estaba programada explícitamente para que la manejara. En la aproximación de estrategias redundantes, el controlador se diseña con estrategias redundantes para realizar una tarea. Existe un modelo de rendimiento para cada estrategia, y una falla se detecta cuando el rendimiento real del comportamiento es peor que el rendimiento esperada. Si la primera estrategia tratada no es suficiente, el controlador eventualmente tratará otra estrategia. El controlador sigue con su repertorio de estrategias hasta que encuentra una con rendimiento aceptable en lugar de tratar

la misma una y otra vez sin éxito. [Pay92] da el ejemplo de un submarino tratando de evitar el lecho marino (Figura 9). Hay varias estrategias que el submarino puede usar para evitar el fondo: elevarse usando los timones, elevarse usando el balasto delantero, incrementar la flotación, y usar la hélice para escapar del fondo. La estrategia preferida es usar los timones. Si el actuador del timón está roto, el submarino no evitará el fondo usando esta estrategia. Después que el controlador determina que el vehículo no está evitando el fondo lo suficiente, se recupera cambiando a la estrategia del balasto delantero. Si el rendimiento todavía no es satisfactorio, el controlador continúa tratando las otras estrategias hasta que el rendimiento sea aceptable.

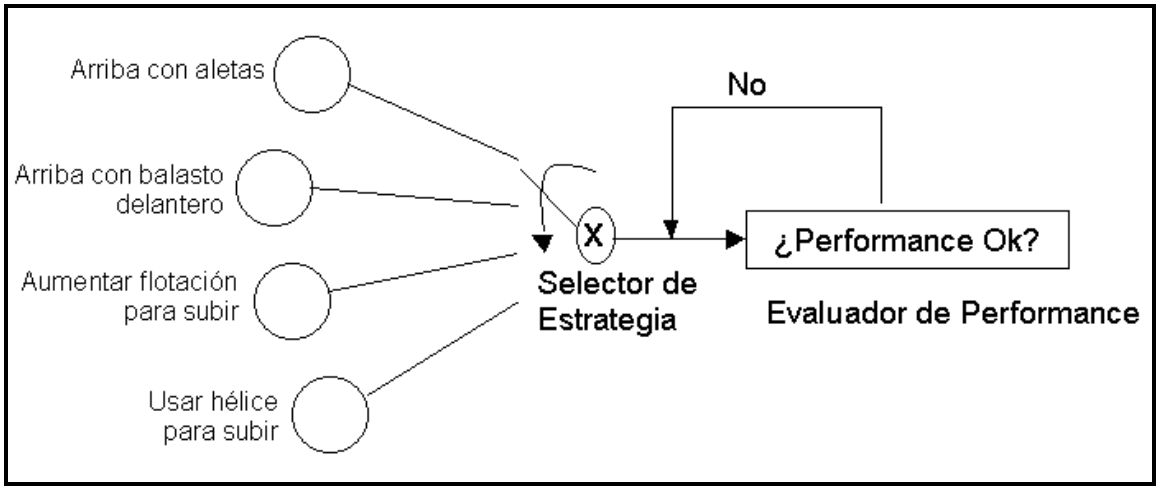

Figura 9 - Comportamiento Robusto con Estrategias Redundantes

Este esquema de redundancia de estrategias no es apropiado para fallas de hardware. Primero, no se dedica específicamente a la causa del problema, sólo se dedica a los síntomas. Puede tomarle varios intentos al controlador encontrar una estrategia que trabaje. Por ejemplo, digamos que Hannibal tiene comportamientos de caminata redundantes; cada comportamiento implementa un paso diferente. Si una pata falla, algunos de estos pasos son inestables. El esquema de estrategias redundantes requiere que Hannibal pase por locomoción inestable hasta que el controlador encuentre un paso que sea estable con la pérdida de esa pata. Es más deseable que Hannibal pudiera reconocer que pata falló y adaptar su paso para dedicarse específicamente a la falla. Segundo, la aproximación de estrategias redundantes requiere que los errores de hardware se manifiesten en el

comportamiento del robot antes de que el sistema de control pueda detectar que algo anda mal. Ergo, el rendimiento de los comportamientos afectados debe degenerar a un nivel inaceptable antes de que el controlador tome una acción correctiva. Esto podría ser perjudicial para un robot que debe funcionar en un entorno peligroso. Tomemos por ejemplo el caso donde un se rompe el sensor usado por un sensor virtual de evitar-un-pozo. El rendimiento del sensor virtual tendría que degradarse lo suficiente antes de que el sistema de control tuviera noticia de la falla. Sería desafortunado si el sistema de control de Hannibal tuviera que esperar hasta que Hannibal pasara por un acantilado antes de que pudiese determinar que el sensor virtual de evitar-un-pozo no está funcionando correctamente. La supervivencia de Hannibal dependería de detectar, enmascarar, y recuperar los errores en el nivel bajo del sistema de control, antes de que los errores afecten los niveles superiores.

SENSORES VIRTUALES ROBUSTOS

El controlador de Hannibal usa sensores virtuales robustos para confinar las fallas de hardware al nivel de control bajo. Los sensores virtuales robustos son sensores virtuales que permanecen confiables a pesar de fallas de sensores reales (Figura 10). Recordar que los sensores virtuales son responsables de caracterizar la interacción del robot con su entorno usando la información de los sensores y de activar los comportamientos del robot. Si los sensores virtuales dan la salida correcta a pesar de fallas de sensores, entonces el robot continuará haciendo las cosas correctas en el momento correcto a pesar de esas fallas.

Por ejemplo, si el sensor virtual de contacto de terreno puede determinar contacto con el terreno a pesar de una falla en el sensor de fuerza, los comportamientos (nivel más alto) que usan información de contacto con el terreno no serán afectados con esta falla.

Los sensores virtuales robustos son interesantes porque confinan el efecto de las fallas al nivel de control bajo y previenen que los errores afecten los niveles superiores. Esta aproximación compensa efectivamente las *fallas locales*. Las fallas locales (también llamadas fallas no catastróficas) son fallas cuyo efecto se confina a la pata que ocurre. Por ejemplo, la falla del sensor del tobillo de una pata

es una falla local porque afecta la habilidad de la pata para medir el contacto con el terreno, pero no afecta la habilidad de otras patas para medirlo.

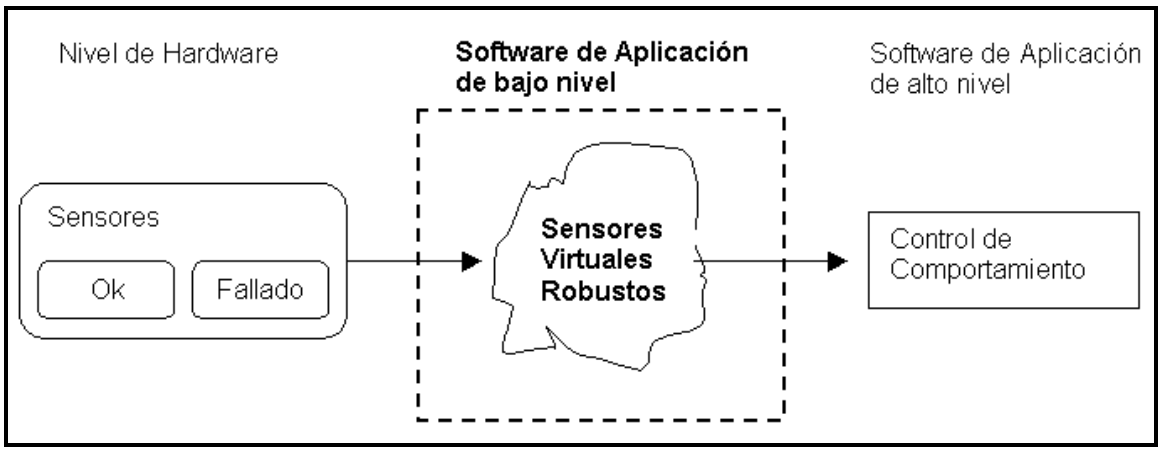

Figura 10 - Sensores Virtuales Robustos

Desgraciadamente no es posible confinar los efectos de todas las fallas de sensores-actuadores al nivel de control bajo. Algunas fallas afectan el comportamiento general del sistema; son llamadas *fallas globales*. Las fallas globales (también llamadas fallas catastróficas) deben compensarse dentro del nivel de control alto. Por ejemplo, si falla el actuador del hombro de una pata, entonces la pata no puede soportar el cuerpo. Así, esta falla afecta la estabilidad global del robot. El nivel de control alto debe compensar esta falla cambiando el paso del robot de modo que pueda caminar de manera estable con una pata menos.

ADAPTACIÓN VS. REDUNDANCIA

Las razones presentadas más arriba hicieron que se elija compensar las fallas locales dentro del nivel de control bajo y compensar las fallas globales en el nivel de control alto. ¿Cómo deberían implementarse los sensores virtuales robustos para dedicarse a fallas locales? ¿Cómo deberían implementarse los sensores virtuales robustos para dedicarse a fallas globales? Una aproximación común es explotar la redundancia para lograr robustez. Para manejar fallas locales, se hubieran podido crear sensores virtuales redundantes para cada interacción pata-terreno de interés para el controlador. Por ejemplo, se podrían diseñar tres

sensores virtuales redundantes de contacto de terreno que usen sensores diferentes para medir contacto con el terreno. Cada sensor virtual redundante de contacto de terreno votaría tanto `contacto=verdarero` o `contacto=falso`, y el veredicto sería por la mayoría de los votos. Para manejar fallas globales, se escribirían agentes redundantes de caminata donde cada agente implementa un paso diferente. El controlador podría activar un agente dado dependiendo de qué patas han fallado. Sin embargo, se encontró que implementar capacidad de tolerancia a fallas usando agentes redundantes era inapropiado para esta aplicación por razones que se exponen más abajo. En su lugar, se usaron agentes adaptables y eso es o que distingue a Hannibal de otras implementaciones tolerantes a fallas.

Inicialmente se usó redundancia para lograr sensores virtuales robustos, pero se abandonó por las siguientes razones. Primero, no hay forma de que el controlador pueda obtener un veredicto confiable una vez que la mayoría de los sensores ha fallado. Segundo, el tamaño del código crece significativamente con este esquema ya que se escriben múltiples agentes de sensores virtuales para cada interacción pata-terreno que se quiera determinar. Es una determinación importante si la memoria de programas disponibles es escasa. Tercero, cuando un sensor real falla, todos los sensores virtuales que usan información de ese sensor se vuelven menos confiables. Así, una minoría de fallas de sensores podría afectar adversamente a una mayoría de sensores virtuales. Payton presenta un ejemplo donde todos los sensores virtuales que activan comportamientos "evitar el fondo" usan un sensor de altitud. Si el sensor de altitud falla, entonces la confiabilidad de todos esos sensores virtuales se degenera. Como consecuencia el rendimiento de evitar el fondo del submarino se degrada con la falla de un sensor. Finalmente, la confiabilidad inherente de los sensores virtuales no es uniforme. Los sensores virtuales que usan más información sensorial producen un resultado más confiable que aquellos que usan menos información. Por ejemplo, digamos que diseñamos tres sensores virtuales de contacto del terreno:

CT_{PFA} el cual usa información de posición, fuerza y tobillo

CT_A el cual usa información de tobillo

CT_F el cual usa información de fuerza

De este modo, CT_{PFA} es más confiable que tanto CT_F como CT_A. Podemos querer reflejar la credibilidad relativa de los sensores virtuales ponderando sus votos de modo que los dos sensores menos confiables no puedan sobrepasar el voto de un sensor significativamente más confiable. Sin embargo, si el sensor de posición falla, entonces CT_{PFA} se vuelve menos confiable que CT_A y CT_F. Repentinamente, queremos que CT_A y CT_F sobrepasen a CT_{PFA}. En esencia, las diferencias entre la credibilidad inherente de los sensores virtuales, y el efecto de los sensores averiados sobre su confiabilidad complica sobremanera el proceso del árbitro. El esquema que usa Hannibal para implementar sensores virtuales robustos es sustancialmente diferente del esquema descripto arriba porque explota la adaptación en lugar de redundancia. Un sensor virtual adaptable (en lugar de varios sensores virtuales robustos) existe por pata para cada interacción pata-terreno de interés para el controlador. Por ejemplo, cuando se detecta una falla, los sensores virtuales apropiados son alertados de la falla y responden volviendo a configurar la forma en que usan su información sensorial. Esto se transmite ignorando la entrada desde el sensor dañado y cambiando la manera en que usan la información de los sensores confiables. De esta forma, los sensores virtuales usan los sensores más confiables para producir el resultado más confiable. Si el sensor dañado vuelve a funcionar nuevamente, el sensor virtual reintegra el componente previamente ignorado.

La aproximación que usa Hannibal para tolerar fallas catastróficas también explota adaptación en lugar de redundancia. Cuando una pata sufre una falla catastrófica no es utilizable. El control de alto nivel debe cambiar el paso de modo que la locomoción permanezca estable con una pata menos. Un esquema redundante podría involucrar implementar comportamientos de caminata redundantes donde cada comportamiento exhibe un paso diferente. Esto no es deseable debido al código extra requerido para implementar cada comportamiento además del mecanismo de cambio de paso. En cambio, el esquema adaptable implementa un comportamiento de caminata el cual puede alterar su paso al cambiar un parámetro. El control de bajo nivel es responsable de detectar fallas catastróficas y

alertar al control de alto nivel; y éste a su vez es responsable de adaptar el comportamiento del robot para que la locomoción permanezca estable.

RED DE TOLERANCIA A FALLAS

Las secciones siguientes describen la red distribuida que implementa la tolerancia a fallas en Hannibal. Como el resto del sistema de control, la tolerancia a fallas se implementa con procesos concurrentes. La tolerancia a fallas consiste en cuatro fases: detección de errores, enmascaramiento, recuperación y reintegración. Las fallas no catastróficas afectan el control local. Se detectan dentro de la red de bajo nivel y se compensan dentro de los sensores virtuales. De esta forma, estas fallas no afectan el rendimiento de alto nivel del sistema. Las fallas catastróficas inevitablemente afectan el rendimiento global del sistema. Se detectan en el control de bajo nivel y se compensan dentro del control de alto nivel.
Se ilustrarán los procesos de tolerancia a fallas de Hannibal con un ejemplo. En el caso de tolerancia a fallas locales se usará un sensor virtual de contacto de terreno. Hay que notar que es el ejemplo de un solo sensor virtual en una pata que usa sólo unos pocos sensores de esa pata. Procesos similares se ejecutan en forma concurrente en la misma pata para hacer robustos también a otros sensores virtuales. El ejemplo para tolerancia a fallas globales se centra en un actuador de hombro dañado. Procesos similares se ejecutan en forma concurrente en la misma pata para tolerar también otras fallas globales. Todos los procesos se implementan en cada pata y se ejecutan simultáneamente.

DETECCIÓN

Los procesos de detección son responsables de reconocer las fallas de sensores y actuadores. La detección es la parte difícil del problema de tolerancia a fallas porque el robot no sabe a priori el comportamiento correcto del sensor. Por ejemplo, si el robot tuviera un mapa preciso del terreno, entonces compararía las lecturas del sensor de su pata con el mapa del terreno. El robot podría esencialmente ir a través de un proceso como “sé que el terreno es plano adelante, de modo que cuando baje la pata los sensores deberían decirme que hice contacto”, o “sé que hay un pozo adelante, de modo que cuando baje la pata

los sensores deberían decirme que hay un pozo". Si el sensor no se comporta como se espera, entonces el robot puede concluir que el sensor está roto.
Sin embargo, Hannibal no sabe como deberían ser las salidas de los sensores para cualquier ciclo de pisada dado. Esto es porque Hannibal cómo es el terreno de antemano. Más aún, Hannibal no puede predecir como será el terreno porque el terreno puede cambiar dramáticamente. ¿Cuándo el robot debería confiar en sus sensores? Hannibal determina la confiabilidad de sus sensores evaluando la salida de sus sensores de la pata en el contexto provisto tanto por la historia de temporal del movimiento de la pata y la salida de sensores complementarios de la pata. Para ilustrar esta idea, imaginar lo siguiente: se despierta en la mañana y quiere saber la temperatura exterior. Miramos el termómetro fuera de la ventana y marca 30℃. Sin embargo, tocando la ventana, ésta está congelada. La mano complementa la lectura del termómetro. Evaluando la confiabilidad del termómetro en el contexto provisto por la historia temporal y los sensores complementarios de la mano, se concluye que el termómetro está roto y se lo ignora. Hannibal emplea el mismo proceso para determinar la confiabilidad de sus sensores.

El reconocimiento de fallas de sensores se realiza usando dos métodos. El primero explota el contexto provisto por la historia temporal del movimiento de la pata. Recordar que el robot no sabe que el comportamiento correcto del sensor es para un ciclo dado del paso. Sin embargo, el robot sabe los movimientos *posibles* de la pata porque dichos movimientos han sido programados para la pata. Al conjunto de los movimientos posibles de la pata se lo llama *modelo*. Si los sensores de la pata reflejan un movimiento posible, es decir, concuerdan con el modelo, entonces el robot tiene cierta confianza de que el sensor está funcionando. Sin embargo, podría darse el caso donde los sensores del robot no reflejen la realidad aunque sí reflejen una realidad posible. Ejemplo: un sensor podría decir que el robot está sobre el piso cuando en realidad está parado en un pozo. Para superar este problema, el segundo método explota el contexto provisto por sensores complementarios. Si los sensores complementarios confiables concuerdan con el sensor en cuestión, (confirman que el robot está pisando el

suelo), entonces el sistema tiene más confianza sobre el sensor. El nivel de confianza en un sensor se refleja por un *parámetro de dolor* asociado al sensor. El nivel de dolor es la función inversa al nivel de confianza.

MODELO DE SENSORES

Es posible modelar el comportamiento del sensor si se conoce el comportamiento de la pata. Los potenciómetros rotativos miden el ángulo de junta de la pata, *strain gauges* miden la carga de la pata y potenciómetros lineales miden la carga del pié. Así, el movimiento de la pata y la interacción de la pata con el entorno afecta directamente la salida del sensor. Para modelar un comportamiento posible del sensor, clasificaremos primero el comportamiento de la pata en términos de estados. Cada fase del ciclo de un paso se divide en cuatro posibles patas.

FALLAS CATASTRÓFICAS

Las fallas globales se detectan en el control de bajo nivel, pero deben ser compensadas en el control de alto nivel. Las fallas de actuadores de hombro, fallas de potenciómetros de hombro, fallas de actuadores de hombro y fallas de potenciómetros de hombro son fallas globales. Estas fallas previene que la pata efectivamente se comporte como estaba programada. Esto es obvio si falla un actuador. El potenciómetro de ángulo de junta falla, los procesadores del servo no tienen forma de conocer el error posicional, de modo que no pueden controlar el actuador. En el evento de una falla global, la pata se señala como no utilizable, y el robot debe modificar su comportamiento para funcionar con menos patas.

DETECCIÓN

Las fallas globales se detectan por los mismos procesos usados para detectar fallas globales. Las fallas de potenciómetros se encuentran usando sus procesos de consenso y monitoreo.

EVALUACIÓN

Para evaluar los aspectos tolerantes a fallas del sistema Hannibal, se tratan los siguientes tópicos:

INTEGRIDAD DE LA DETECCIÓN DE FALLAS

El sistema detecta exitosamente un gran espectro de fallas comunes. Distingue entre fallas catastróficas y no catastróficas. Respecto de las fallas locales, reconoce el tipo de sensor que ha fallado. Respecto de las fallas globales, reconoce fallas de potenciómetros. Las fallas de actuadores aparecen como la falla masiva de todos los sensores cuyo comportamiento depende de ese actuador. Cuando suceden este tipo de fallas catastróficas, los correspondientes potenciómetros aparecen como rotos. De este modo, todas las fallas catastróficas evocan el mismo procedimiento de recuperación. Así, el sistema sólo busca fallas de potenciómetros para detectar fallas catastróficas.

COBERTURA DE FALLAS

El robot se recupera con éxito de un gran espectro de fallas, tales como fallas de sensores, fallas de actuadores y fallas de procesadores locales. Identifica fallas con un efecto local tanto como con efecto global. Maneja errores transitorios, descalibración de sensores y fallas permanentes. Compensa varias combinaciones de fallas: fallas que ocurren individualmente, en concurrencia con otras fallas o acumuladas con el tiempo. Tolera estas combinaciones de fallas en varias permutaciones.

CONFINAMIENTO DE ERRORES

El sistema previene en forma efectiva fallas de sensores y actuadores de influenciar en forma adversa el comportamiento del robot. Para lograr esto, los procesos de detección monitorean las salidas de los sensores y alerta de las fallas que puedan aparecer. Esto corta de raíz el problema de la falla porque el sistema puede compensar efectivamente las fallas una vez que conoce cuándo y qué tipo de fallas han ocurrido. Por ejemplo, los sensores virtuales robustos filtran los efectos de las fallas no catastróficas de modo que estas fallas no influencien el comportamiento del robot.

TIEMPO DE RESPUESTA A FALLAS

El sistema detecta y se recupera con éxito de fallas antes que el rendimiento del robot se degrade a un nivel inaceptable. Un tiempo de respuesta rápido es importante para implementar correctamente el confinamiento de errores. Después de todo, le ayuda de nada al sistema implementar los procedimientos de detección y recuperación en el control de bajo nivel, si le toma demasiado tiempo para responder a las fallas.

EXTENSIÓN DE LA DEGRADACIÓN GRADUAL DEL RENDIMIENTO

Los procesos de recuperación mantienen el rendimiento del robot al nivel más alto posible dado el estado funcional del hardware. Ya que el sistema reconoce las fallas, los procesos de recuperación pueden acomodar el uso de los sensores y actuadores para minimizar los efectos de las fallas en el comportamiento del robot. Al nivel de los sensores virtuales, los procesos de recuperación sopesan velocidad por confiabilidad a medida que disminuye el número de sensores funcionales. Al nivel de locomoción, los procesos de recuperación sopesan velocidad por estabilidad a medida que disminuye el número de patas útiles.

DISPONIBILIDAD DE RECURSOS CONFIABLES

Los procesos de reintegración reincorporan los componentes reparados de modo que el robot tenga acceso a todos los recursos confiables. Esto es importante porque cuantos más sensores y actuadores pueda usar el sistema, mejor será su rendimiento. El tiempo de respuesta de reintegración es relativamente rápido de modo que el sistema no tiene que esperar mucho antes de poder volver a usar los componentes reparados.

DIVISIÓN DE LAS RESPONSABILIDADES DE TOLERANCIA A FALLAS ENTRE HARDWARE Y SOFTWARE

El sistema implementa todas las capacidades de tolerancia a fallas dentro del software. Se eligió no implementarlas al nivel del hardware por razones de costo y peso. En consecuencia, el código se incrementó bastante. Se estima que los procesos de tolerancia a fallas requieren la misma cantidad de código que el código de locomoción y el código de terreno abrupto combinados. En otros

sistemas podría tener sentido sopesar entre el tamaño del código y el peso y costo del hardware.

COMPARACIÓN CON OTROS SISTEMAS

La tolerancia a fallas es un área relativamente no explorada en la investigación de robots autónomos. En esta sección, se compara la aproximación de Hannibal con otros trabajos en este campo.

REPLICACIÓN DE HARDWARE

La replicación de hardware se usa comúnmente para mejorar la confiabilidad del hardware. El sistema se hace tolerante para un tipo particular de componente replicándolo e ignorando los componentes que exhiban comportamiento en minoría. La duplicación de hardware/software aumente la tolerancia a fallas del sistema, pero no necesariamente provee capacidades mejoradas sobre el sistema no replicado. En contraste, este esquema tiene la ventaja de usar componentes con capacidades complementarias y a la vez diferentes para lograr tolerancia a fallas. Ya que los componentes tienen capacidades complementarias, pueden usarse para propósitos diferentes. De este modo los componentes adicionales no solo aumentan las capacidades de tolerancia a fallas del sistema, sino que proveen capacidades adicionales y de rendimiento más allá de lo que el sistema tuviera con menos componentes.

RESULTADOS

El comportamiento tolerante a fallas es importante para robots móviles autónomos independientemente de su tarea. Construir robots móviles autónomos capaces de realizar tareas en entornos demasiado peligrosos o no aceptables para humanos es un objetivo de la robótica y una rama con variadas aplicaciones. Un ejemplo de ello es la exploración planetaria con micro-exploradores. Estas tareas requieren que el robot lleve a cabo su misión durante largos períodos sin darse el lujo de una reparación cuando un componente falla. Es vital para el éxito de la misión que el robot siga funcionando a pesar de fallas en sus componentes. Ya que la principal ventaja de un robot autónomo es su habilidad para realizar tareas sin la asistencia

o intervención humana, es sorprendente que el tema tolerancia a fallas permanezca relativamente inexplorado. La tolerancia a fallas debe investigarse con profundidad a medida que se vuelve más importante para los robots autónomos permanecer largo tiempo sin reparaciones.

Con Hannibal se ha demostrado que un robot autónomo puede detectar de forma confiable fallas de sensores comparando el comportamiento real del sensor con su comportamiento esperado y con el comportamiento de sensores complementarios. Esto tiene dos implicaciones importantes. Primero, si el sistema puede reconocer cuando los componentes han fallado, puede ajustar el uso de los restantes sensores y actuadores para minimizar el impacto de la falla en el rendimiento del robot. Segundo, si el robot puede reconoce cuando los componentes están funcionando, entonces puede integrar el uso de todos los componentes operativos para mejorar el rendimiento del robot. Esto es útil para reintegrar componentes reparados. Ambas capacidades se han demostrado en Hannibal.

La tolerancia a fallas en robots autónomos es un área de investigación llena de posibilidades para trabajos futuros. Se ha demostrado las capacidades de tolerancia a fallas iniciales en Hannibal, pero hay mucho terreno para mejorar. Los comportamientos de los sensores están fijos en el código (*hard-wired*). Como resultado, el robot sólo es capaz de detectar confiablemente fallas en entornos para las cuales tiene patrones. Por ejemplo, los valores de los sensores podrían parecer diferentes de lo esperado si el robot camina en el barro. Un avance lógico es hacer que el robot aprenda nuevos patrones a lo largo del tiempo para poder reconocer fallas de sensores en nuevos entornos. Esta es un área muy ligada a la Inteligencia Artificial. También se puede trabajar sobre como implementar capacidades de tolerancia a fallas con diferentes clases de sensores y distintos números de sensores. Sin embargo no debe confundirse con el concepto *integración de sensores*, ya que bajo esos términos se están desarrollando tareas concernientes a proveer a equipos militares las capacidades de obtener

información del teatro de operaciones de la forma más segura y exacta posible y su aplicación a armas (específicamente aviones de combates y misiles).

Tolerando Fallas de Sensores Continuamente Valuados

Los programas de control de procesos deben tener una característica especial: la habilidad para tolerar fallas de sensores. Según [Mar90] se presentará una metodología para transformar un programa de control de procesos que no puede tolerar fallas de sensores en uno que sí puede. Se modifican las especificaciones para acomodar incertidumbre en los valores del sensor y el sensado promedio de una forma tolerante a fallas. Además se presenta una jerarquía de modelos de fallas de sensores. Un programa de control de proceso se comunica y sincroniza con un proceso físico. Típicamente, el programa lee valores del proceso físico a través de sensores y escribe valores por medio de actuadores, según el esquema de la Figura 11:

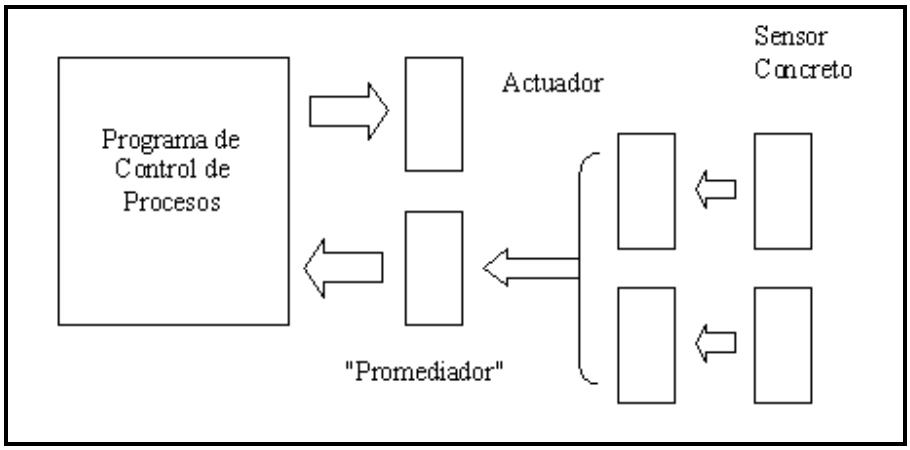

Figura 11 - Esquema básico

La aproximación usada en este trabajo se describe como sigue:

1. Se escribe una especificación del programa de control en términos de las variables de estado del sistema físico. Por ejemplo, la especificación de un programa que controla un tanque de reacción química se referirá a una variable T cuyo valor se asume que es la temperatura del tanque.

2. Cada variable física de estado referenciada por la especificación se reemplaza con una referencia a un *sensor abstracto*. Un sensor abstracto es un conjunto de

valores que contiene la variable física de interés. Aparece entonces la incertidumbre de los valores del sensor, y debe reexaminarse la especificación para acomodarla.

3. Se escribe el programa de control basado en la especificación producida por el paso 2. Este programa lee valores abstractos que se asumen contienen siempre el valor correcto de las correspondientes variables físicas.

4. Para cada sensor abstracto referenciado en por el programa del paso 3, se construye un conjunto de sensores abstractos que fallen independientemente. Cada sensor abstracto se implementa usando un *sensor concreto*, un dispositivo físico que lee una variable física, como ser un termómetro (el sensor concreto no necesita sensar la variable física de interés; por ejemplo, un sensor abstracto de temperatura podría construirse a partir de un manómetro usando la ley de Boyle: PV = nRT). Este paso requerirá cierto conocimiento del proceso físico a ser controlado como también las características del sensor concreto.

5. Se usa un algoritmo promediador (averaging algotihm) tolerante a fallas con los valores de estos sensores abstractos replicados, para calcular otro sensor abstracto que es correcto aún si algunos de los sensores originales son incorrectos.

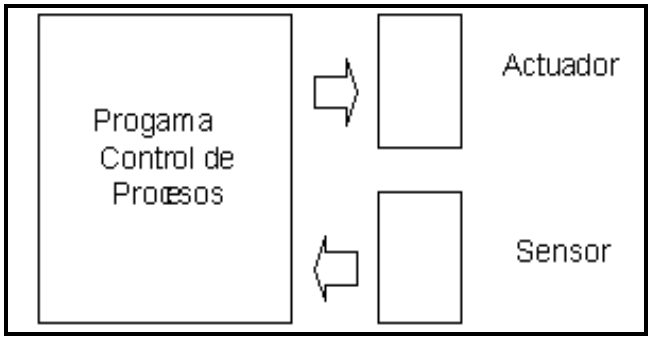

Figura 12 - Esquema del Sistema Resultante

El algoritmo asume que no más de f de los n sensores abstractos son incorrectos, donde f es un parámetro. La relación entre f y n dependerá de la forma en que los sensores puedan fallar. El sistema resultante tendrá la estructura presentada en la Figura 12.

VARIABLES FÍSICAS DE ESTADO Y SENSORES CONCRETOS

Una variable en una computadora es diferente de una variable de estado en un proceso físico. Una variable en una computadora toma valores de un dominio finito, y puede asumir sólo un limitado número de valores en cualquier período finito de tiempo. Una variable física de estado, sin embargo, puede tomar cualquier valor real en momentos arbitrarios. Una forma conveniente de representar una variable física de estado en una programa de computación es mediante una función. El dominio de tal función es típicamente el tiempo, pero puede ser otra variable.

Un sensor concreto es un dispositivo que puede usarse para hacer un muestreo una variable de estado físico. Del ejemplo anterior del tanque, el sensor concreto podría ser un termómetro. Un sensor concreto puede interactuar con la computadora de muchas formas:

- la computadora puede encuestar (*polling*) al sensor
- el sensor puede alertar asincrónicamente a la computadora de que cierto valor es sensado
- el sensor puede enviar a la computadora una corriente de valores donde cada valor indica que la variable física a cambiado por una cierta cantidad

Se asume que el sensor tiene una especificación y se lo considera averiado si exhibe un comportamiento no consistente con su especificación.

Un sensor concreto no es un mecanismo muy conveniente. El ejemplo del termómetro:

- El sensor tiene exactitud limitada. Demoras en la red y la planificación del procesador limitan aún más la exactitud del sensor.
- El programa de control puede estar interesado en una temperatura en un momento en que no es hecho un muestreo del termómetro. Debe interpolarse un entonces un valor; hacerlo requiere conocimiento del proceso físico que se está controlando.
- Algunas propiedades del sensor concreto, mientras que son importantes para la implementación, pueden ser irrelevantes para la especificación usada por el programa de control del proceso. Otro termómetro podría generar una interrupción

si la temperatura supera 100 ℃. Esta es una propie dad importante del sensor, permite una determinación exacta de cuando se alcanzan 100 ℃. Habría otras formas de hacer la misma clase de medida precisa, sin embargo, para un sensor que es encuestado (*polled*). Sería conveniente si el programa de control pudiera ser el mismo para cualquier método de medición, siempre que sea lo suficientemente exacto.

Los problemas de interpolación y abstracción de datos se tratan por medio de sensores abstractos. La falla de un sensor abstracto puede sobrevenir si falla el sensor concreto subyacente. Se define una jerarquía de clases de fallas definida como:

- Fallas *fail-stop*, donde puede detectarse un sensor abstracto que ha fallado.
- Fallas arbitrarias con inexactitud limitada, donde pueden detectarse sensores abstractos que son demasiado inexactos.
- Fallas arbitrarias, donde un sensor abstracto puede fallar arbitrariamente.

Dado un sensor concreto, puede ser difícil implementar un sensor abstracto. En general, requiere cierto conocimiento del proceso físico que se está controlando.

SENSORES ABSTRACTOS TOLERANTES A FALLAS

Dados *n* sensores abstractos independientes y ciertas suposiciones acerca de fallas, se quiere construir un sensor abstracto que sea tolerante a fallas. Primero se presenta un algoritmo para construir un sensor que contenga el valor correcto, dado que no más de *f* de los sensores originales no sean correctos. Una serie de valores sirven para determinar el grado de replicación de sensores dependiendo de la cantidad de sensores defectuosos sobre el modelo que se use.

Tolerancia a Fallas en un Entorno Multisensor

La replicación de sensores es deseable no sólo para tolerar fallas de sensores sino para incrementar la exactitud esperada de un conjunto de sensores replicados más allá de la que se puede obtener con un único sensor. La replicación de este tipo de componentes da lugar a la aparición de entornos multisensor o la formación de una red distribuida de sensores.

En [Jay94] se modela un sensor continuamente valuado como un intervalo de números reales que contiene el valor físico de interés. Dados *n* sensores de los cuales *f* pueden sufrir fallas arbitrarias, se presenta un algoritmo cuya salida es confiable, es decir, garantiza que contiene el valor correcto en todo momento, y es lo suficientemente correcto cuando $f < n/2$.

Los datos que surgen de un sensor pueden ser defectuosos por fallas en el sensor o ruido en el ambiente. Si se ha usado replicación de sensores, se necesita un método para combinar la información de los distintos sensores, y en la literatura esta acción se denomina integración de la información, la cual puede ser *competitiva* o *complementaria*. Para el caso primero, cada sensor brinda, en teoría, idéntica información, y como ello no sucede en la práctica, se necesita replicación de sensores. En el caso de la complementación, la integración se realiza ya que cada sensor aporta sólo datos parciales, entonces requiriendo que se combine tal información para obtener el conocimiento deseado. El objeto del trabajo se centra en la integración de información competitiva, es decir, integrar los datos de diferentes sensores para poder descartar aquellos defectuosos y seguir siendo capaces de disponer del dato necesario. Utilizando como base [Mar90] (trabajo que se discute en el título anterior TOLERANDO FALLAS DE SENSORES CONTINUAMENTE VALUADOS), se hacen distinciones entre un *sensor concreto*, un *sensor abstracto* y un *sensor abstracto confiable*. Un sensor concreto es un dispositivo que hace un muestreo de la variable de estado física de interés. Un sensor abstracto A es una función continua que va de la variable de estado física a un intervalo de números reales $[l_a, u_a]$ (con $l_a < u_a$).
Estos conceptos posibilitan entonces las siguientes definiciones:

- El *ancho* de un sensor abstracto A es $(u_a - l_a)$. El ancho determina la exactitud de un sensor. Cuanto menor el ancho, mejor la exactitud.
- Un sensor abstracto A es *más exacto* que un sensor abstracto B si $(u_a - l_a) < (u_b - l_b)$.

- Un sensor abstracto *correcto* es un sensor abstracto cuyo intervalo contiene el valor físico presente y cuyo ancho es al menos Γ (donde Γ es un parámetro especificado por el diseñador).
- Un intervalo A se solapa completamente con un intervalo B si y sólo si $l_a <= l_b$ y $u_a \geq u_b$.
- El ancho de un sensor abstracto *confiable* es el ancho de su intervalo de extensión.

Estas definiciones se representan a continuación (Figura 13):

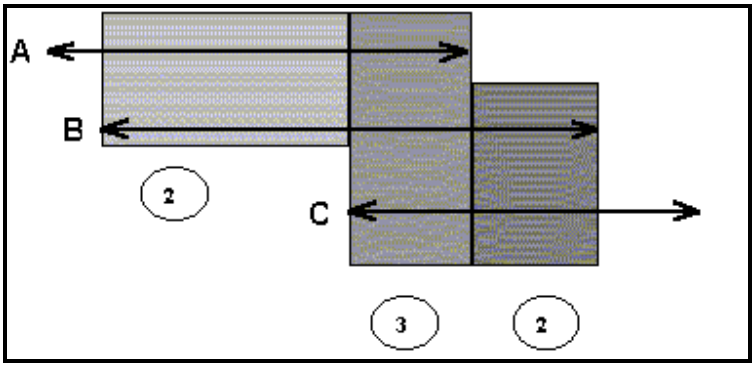

Figura 13 - A, B, C tres sensores abstractos

El resultado del trabajo es un algoritmo que mejora el de [Mar90] ya que no sólo provee un único intervalo como en ese caso, sino que además detecta todos los posibles sensores defectuosos (el algoritmo de Marzullo puede dejar de detectar alguno de ellos). Esta línea de trabajo puede extenderse para incorporar sensores multidimensionales, al reemplazar cada intervalo correspondiente a un valor físico por un vector de intervalos.

Algoritmos de Sensado Robusto y Distribuido

Una de las tácticas de tolerancia a fallas para evitar que los sistemas sean vulnerables a la falla de un solo componente, es razonable el uso redundante de varios sensores. Un sistema de seguimiento automático podría usar diferentes tipos de sensores (radar, infrarrojo, microondas) que no son vulnerables a las mismas clases de interferencias. Ahora bien, la solución aparente de la redundancia también trae aparejada problemas ya que el sistema recibirá varias lecturas que son parciales o enteramente erróneas. El sistema deberá ser capaz

de decidir que componentes están fallados, como también como interpretar al menos lecturas parcialmente contradictorias.

Para mejorar la confiabilidad de un sistema que usa sensores, se ha estudiado en [Bro96] el problema práctico de combinar, o *fusionar*, los datos de muchos sensores independientes en una lectura confiable. Al integrar las lecturas de los sensores, se debe cuidar la robustez y la confiabilidad. La cuestión pasa por poder tomar la decisión correcta en la presencia de datos erróneos.

PRECISIÓN Y EXACTITUD

Mucho dependerá de la *exactitud* (la distancia entre los resultados y los resultados deseados) y la precisión (el tamaño del rango de valores retornados) de los sensores empleados por el sistema (Figura 14).

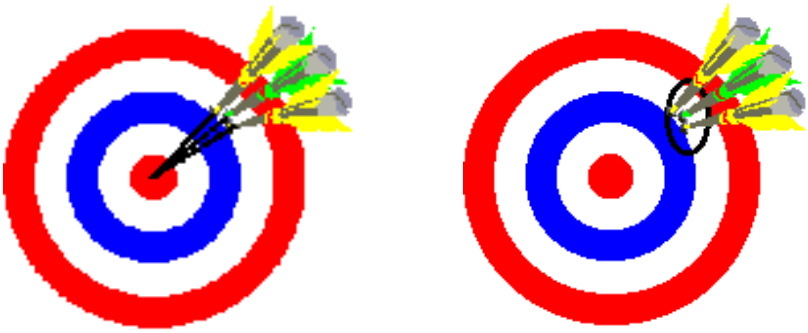

Figura 14 – Diferencia entre Exactitud y Precisión

Ya que los sensores utilizados en sistemas de computación son inherentemente no confiables, al distribuir los sensores se compromete aún más la confiabilidad de los mismos. Existen algoritmos de fusión de sensores y acuerdo bizantino. Dichos algoritmos (de acuerdo aproximado y fusión de sensores) aceptan datos de muchas fuentes independientes, donde una minoría de los datos son erróneos (ya sea por carecer de exactitud o de precisión). Más aún, al emplear sensores en un sistema de tiempo real, es muy importante distinguir que las lecturas de los sensores son útiles sólo si son correctas y el rango entre ellas (la diferencia entre los valores aportados por los distintos sensores usados) es lo suficientemente pequeña.

Una serie de algoritmos son revisados y comparados a partir de un conjunto único de valores, que se aplican a cada instancia de los ejemplos para luego resumir los resultados obtenidos con cada uno de ellos. Además, el trabajo presenta su propio algoritmo, el cual combina dos de los algoritmos presentados, de esa manera logrando mejorar la exactitud e incrementar la precisión obtenida hasta ese momento. Al basar la solución en dos componentes derivadas de teorías independientes (una de la teoría de conjuntos, la otra de geometría), se producen dos explicaciones distintas al mismo problema, lo cual hace que la solución sea más fácil de entender.

Resumen

Dentro de esta sección se han tratado diversas características de tolerancia a fallas (discutidas en la sección TOLERANCIA A FALLAS) con su aplicación directa en distintas áreas de los sistemas de tiempo real. Se seleccionaron las áreas de diseño y especificación, la planificación de tareas y el control de sensores y actuadores. Se ha presentado un panorama ciertamente completo respecto del estado del arte en cada una de esas áreas. El estudio de la gran cantidad de trabajos relevados y consultados permite concluir que la brecha entre la tolerancia a fallas y los sistemas de tiempo real como campos de investigación independientes se ha reducido notablemente en los últimos años. El empleo cada vez más amplio de sistemas de tiempo real implica que necesariamente deban ser tolerantes a fallas, de allí la necesidad del desarrollo de técnicas específicas o la adopción de soluciones ya disponibles pero condicionadas a las características particulares de los sistemas de tiempo real.

La disponibilidad de lenguajes para especificación y diseño con extensiones de tolerancia a fallas brinda la oportunidad de incluir esas características desde la concepción de los sistemas, reduciendo así los problemas encontrados en otras épocas al tener que reformular una especificación para acomodar técnicas de tolerancia a fallas, y permitiendo la verificación formal de dichas especificaciones antes de pasar a la etapa de implementación.

La planificación de tareas, una de las áreas de mayor investigación con relación a su incorporación a sistemas operativos de tiempo real, muestra la diversidad de algoritmos que incluyen capacidad de tolerancia a fallas, ya sea a partir de la variación de algoritmos existentes o probados, o mediante ideas noveles. Esto hace que el diseñador de un sistema de tiempo real pueda esperar que esos servicios estén disponibles a través de llamadas al sistema operativo, y de esta forma disminuir el tiempo destinado a escribir sistemas tolerantes a fallas.

La profusión de sensores y actuadores, sobre todo en el desarrollo de sistemas autónomos (como aquellos empleados en la exploración espacial) hace de la idea de redundancia un concepto clave para la tolerancia a fallas, especialmente por los entornos en los que son empleados dichos sistemas. Pero a la vez, trae aparejada la aparición de problemas a tratar, como ser la integración de la información redundante o la detección de componentes defectuosos. Es esta última área la elegida para ampliar el tema; la próxima sección mostrará, a través de experiencias con un modelo real, la aplicación de algoritmos de sensado probados en un sistema operativo con extensiones de tiempo real.

4. Experiencias con un Modelo Real

"Ver para creer"

Santo Tomás de Aquino

Objetivo

Los conceptos presentados hasta el momento en los capítulos precedentes, tienen un gran componente teórico. Sin embargo, la realidad impone en muchas ocasiones condiciones distintas a las ideales, cuando esas condiciones fueron enunciadas. La idea es tomar algunos de los temas tratados anteriormente y probar soluciones en un entorno real para demostrar su validez. Para ello, se implementó un modelo de sensores replicados, usando como plataforma el sistema operativo MINIX con extensiones de tiempo real (RT-MINIX de aquí en más), según lo propuesto en [Wai95] y extendido de acuerdo a [Rog99].

Las soluciones propuestas se implementaron primero dentro de programas de usuario, y las que presentaron mejor comportamiento, han sido incorporadas luego a dicho sistema operativo. Mediante este esquema se pudieron atacar tres puntos al mismo tiempo:

- Hacer extensiones sobre tolerancia a fallas mediante replicación, ya sea tanto física (sensores y/o actuadores en el modelo) como lógica (tareas en el sistema operativo).
- Probar a RT-MINIX como una base confiable para desarrollos reales.
- Comprobar en forma práctica, a través del modelo, los alcances de características de tolerancia a fallas en una aplicación de tiempo real.

Extensiones de Tiempo Real a MINIX

Consideraciones Generales

MINIX [Tan87] (nombre que significa *MI*ni-U*NIX*) es un sistema operativo completo, con capacidades multitarea y multiusuario. Desarrollado a semejanza de

UNIX, fue creado por A. Tanenbaum en la Universidad Vrije de Amsterdam, Holanda, con el propósito de que sirviera para enseñar e investigar conceptos de sistemas operativos. Aún cuando el código fuente es propiedad de los autores, se ha puesto ampliamente a disposición de instituciones académicas, y en muchas de ellas ha sido utilizado con fines educativos y de investigación.

[Wai95] mostró los resultados obtenidos en un proyecto de investigación destinado a usar MINIX para implementar planificación de tiempo real. Varios cambios al código fuente del *kernel* de la versión 1.5, permitieron al programador contar con un conjunto de nuevas llamadas al sistema operativo para la creación y manejo de tareas, tanto periódicas como aperiódicas. También se incluyeron otros servicios de tiempo real, como ser la planificación RMS y EDF. La disponibilidad de esta herramienta permitió realizar nuevos trabajos variando desde pruebas de nuevos algoritmos de planificación o la identificación de características adicionales que debían ser agregadas como modificaciones al *kernel*. Mientras tanto, la aparición de nuevas versiones de MINIX (1.7 y 2.0) creó la necesidad de integrar el trabajo previo junto a nuevos servicios adicionales necesarios para el propósito de esta tesis en una nueva versión de RT-MINIX.

[Rog99] describe el trabajo realizado para transformar MINIX 2.0 en RT-MINIX. Se incluyeron los servicios de tiempo real disponibles en la versión anterior de RT-MINIX y se incorporaron tres características importantes. La primera es totalmente nueva, ya que se dotó a RT-MINIX de la posibilidad de acceder a la palanca de juegos (*joystick port*) de una PC (permitiendo de esta forma acceder a los conversores A/D provistos en la misma) mediante un nuevo controlador de dispositivos, facilitando la conexión de diversos sensores y ampliando la gama de posibles elementos a conectar, a los existentes puertos serie y paralelo. Además, se trabajó en mejorar la disponibilidad de valores estadísticos respecto de la actividad del sistema operativo de tiempo real, como se cantidad de tareas creadas y activas, metas perdidas y algoritmo de planificación en uso. Si bien estos valores eran accesibles por el programador en la versión anterior, había una

gran diversidad de llamadas al sistema operativo. Se optó entonces por reunir los datos estadísticos bajo una sola llamada, logrando así facilitar su uso. Por último, se adoptó lo propuesto en [WCG98] (probado sobre MINIX 1.5) relativo a la unificación de las colas de planificación usadas por MINIX. Esta aproximación demostró ser factible también sobre MINIX 2.0 y se consiguió mejorar el tiempo de respuesta del sistema operativo ya que la unión de las colas (pasando de las tres originales a sólo dos: tareas del sistema y tareas de usuario) reduce la interferencia de las tareas del sistema operativo con la mayoría de las tareas de tiempo real. De esta manera se dio un paso hacia la tolerancia a fallas de RT-MINIX. Por último, también se incorporó al sistema operativo (luego de pruebas con programas de usuario) un servicio para aplicar algoritmos de sensado robusto a un conjunto de lecturas de sensores; este tema se relaciona con uno de los puntos expuestos anteriormente en el capítulo Estado del Arte, y se trata con profundidad en Algoritmos de Sensado Robusto más adelante.

Partiendo desde una instalación MINIX 2.0 básica, se trabajó en portar todo lo disponible en RT-MINIX anterior (basado en una instalación MINIX 1.5). En los casos de servicios existentes en la versión anterior, la tarea consistió en estudiar primero las diferencias (en caso de existir) entre los programas de MINIX 1.5 y 2.0 en los cuales se efectuarían los cambios al código fuente para entender el contexto y luego comprender el propósito e impacto de dichos cambios. Para los desarrollos completamente nuevos (por ejemplo el controlador de dispositivos, los algoritmos de sensado robusto y los programas de ejemplo), hubo libertad a la hora crear nuevos programas si era necesario, pero también se debió considerar el impacto de estos agregados en la estructura existente. Esta etapa implicó realizar una gran actividad de planificación y estudio antes de pasar a la codificación propiamente dicha.

Cada extensión se trabajó por separado, es decir, sólo se avanzó en incorporar una nueva característica luego de que la anterior estuviera debidamente probada y fuera considerada estable. En los párrafos que siguen a continuación, se

describen en detalle las tres extensiones de tiempo real más importantes realizadas al sistema operativo MINIX 2.0 en relación con el propósito de esta tesis; extensiones no disponibles en versiones anteriores y que son el resultado del presente trabajo. Como subproducto se obtuvo además un paquete distribuible de RT-MINIX, un conjunto de programas que aplicados a una instalación MINIX 2.0 básica permiten transformarla, luego de un proceso de compilación, en un sistema operativo de tiempo real: RT-MINIX.

Controlador de Dispositivos para la Palanca de Juegos - Conversor A/D

Como se mencionó en la sección Conceptos, una característica distintiva en un sistema de tiempo real es su interacción con el medio ambiente, ya que la mayoría de ellos son usados para controlar procesos reales. Para ello se emplean sensores, que interactúan con el entorno mismo y permiten conocer su estado. La lista de sensores disponibles es muy larga (temperatura, presión, infrarrojo, sonar, microondas, etc.) muchos de ellos proveyendo señales analógicas como forma de respuesta.

La interface del puerto de juegos en una PC permite conectar hasta cuatro entradas analógicas y cuatro entradas digitales (generalmente en la forma de dos palancas de juegos). Capacitar al sistema operativo con la habilidad de leer este puerto amplía las opciones de conectar diferentes sensores analógicos, una opción muy interesante para aplicaciones de tiempo real. Ninguna de las versiones de MINIX originales proveía la posibilidad de acceder a estos dispositivos, por lo tanto fue necesario escribir un controlador de dispositivos para este fin.

La palanca de juegos de la PC puede emplearse como un conversor A/D de cuatro canales. Las entradas resistivas (coordenadas XY) y las entradas digitales (pulsadores) se alinean juntas en un byte (8 bits) que puede leerse en la dirección de memoria 0x201. La correlación de los pines del conector-D con los bits del bus de datos se aprecia en la Figura 15.

El principio de funcionamiento es escribir cualquier valor en el puerto 0x201h y realizar un ciclo de lectura para ver cuánto tarda en volver a cero cada una de las cuatro entradas resistivas. El número de veces que se ejecuta el ciclo será directamente proporcional a la resistencia de esa entrada (ya sea la posición de la palanca de juegos o el valor de un sensor conectado a la misma). El estado de un pulsador (abierto o cerrado, en este caso es una entrada digital) se conoce consultando el nibble más alto de la dirección 0x201. Si uno de los bits es 0 (cero), el pulsador correspondiente ha sido apretado. Por analogía, un 1 (uno) en esa posición significará que el botón no está pulsado (el circuito está abierto). Este tipo de entrada puede ser de utilidad para conectar sensores del tipo lógico (por ejemplo limitadores de carrera o interruptores de posición).

Entradas	***Dirección 0x201***	***Pines***	***Función***	***Palanca***
Digitales	Bit 7	14	Botón 2	B
	Bit 6	10	Botón 1	
	Bit 5	7	Botón 2	A
	Bit 4	2	Botón 1	
Resistivas	Bit 3	13	Coord. Y	B
	Bit 2	11	Coord. X	
	Bit 1	6	Coord. Y	A
	Bit 0	3	Coord. X	

Figura 15 - Correspondencia entre pines y bits del bus de datos

Los distintos controladores de dispositivos encontrados en MINIX 2.0 no sirvieron de orientación en la forma de encarar la estructura de este nuevo controlador (si bien básicamente todos son una tarea de E/S, los disponibles se relacionaban con el reloj interno, un disco rígido, o una placa de red, y no daban pistas para seguir). El primero escrito a tal fin, investigado en [PRRS96] no brindaba la resolución apropiada para leer los sensores conectados a este puerto, ya que dejaba en manos del programador la tarea de realizar el ciclo de lectura. Esto provocaba demasiada demora, pues se efectuaba a nivel tarea de usuario, y los resultados logrados en los contadores eran bajos. La amplitud (o diferencia entre el menor y mayor de los valores leídos) era realmente poco apreciable.

Para llevar a cabo esta extensión se recurrió al esqueleto encontrado en un controlador de dispositivos escrito para el sistema operativo Linux según [Pau98], con diversas modificaciones (tanto para eliminar las referencias específicas al sistema operativo original, como para simplificar ciertas partes del código, que nada tenían que ver con la operación de E/S propiamente dicha, sino más bien con cuestiones cosméticas para facilitar su uso en programas de juegos y que no se aplicarían en esta tesis).

```
io_task()
{
   int r, caller;
   message mess;               /* message buffer */

   initialize();               /* only done one at system init */
   while (TRUE) {
      receive(ANY, &mess);     /* wait for a request for work */
      caller = mess.m_source;  /* process sending the message */
      switch(mess.m_type) {    /* handle each request type */
         case READ:   r = do_read(); break;
         case WRITE:  r = do_write(); break;
         case OTHER:  r = do_other(); break;
         default:     r = ERROR;
      }
      mess.m_type = TASK_REPLY;
      mess.REP_STATUS = r;     /* result code */
      send(caller, &mess);     /* send reply message to caller */
     }
}
```

Programa 1 - Cuerpo de una rutina de E/S en MINIX [Tan92]

El cuerpo de una tarea de entrada/salida se muestra en el Programa 1, según [Tan92]. Todas las tareas de este tipo se deben escribir para recibir un mensaje, llevar a cabo una acción apropiada y devolver una respuesta.

Una tarea de entrada salida puede dividirse en dos partes: una independiente y otra dependiente del dispositivo a controlar, teniendo que poder proporcionar cuatro acciones básicas: apertura, lectura, escritura y cierre de dicho dispositivo. Esa correlación se ejemplifica en la Figura 16.

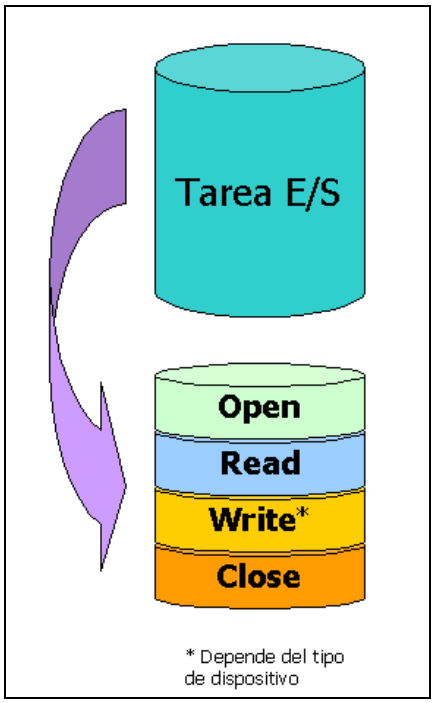

Figura 16 - Acciones que debe proveer una tarea de E/S en MINIX

Este servicio incorpora una nueva tarea en el *kernel* que provee al programador tres operaciones básicas (abrir, leer, cerrar) para acceder al puerto de juegos como dispositivos de caracteres (`/dev/js0` y `/dev/js1`, para la palanca A y la palanca B, respectivamente); dispositivos no existentes en una distribución MINIX básica y que fueron creados específicamente durante este trabajo.

Se agregaron los archivos `/usr/src/kernel/joystick.c` y `/usr/include/minix/joystick.h`. Se crea la nueva tarea GAME_TASK (a semejanza de las originales, como PRINTER_TASK o CLOCK_TASK) encargada de atender los pedidos de apertura, lectura y cierre del dispositivo en la forma de mensajes al kernel. Una vez recibido el mensaje, se invoca la función que corresponda según se pida abrir, leer o cerrar el dispositivo. En el primer caso, se efectúan los controles necesarios para comprobar que el dispositivo pedido sea correcto, que exista conectado un joystick/sensor y que no esté abierto por otro proceso. Se devuelve el descriptor de archivo (*file handle*) o el código de error si ese fuese el caso.

```
int js_read(m_ptr)
register message *m_ptr;   /* pointer to message from user */
{
  int t_x0, t_y0, t_x1, t_y1, t0, j, buttons, minor, chk;

  minor = m_ptr->DEVICE;
  buttons = ~(in_byte(JS_PORT) >> 4);
  js_data[0].js_save.buttons = buttons & 0x03;
  js_data[1].js_save.buttons = (buttons >> 2) & 0x03;
  /* Disable interrupts until port has been read */
  lock();
  /* initialize counters */
  t_x0 = t_y0 = t_x1 = t_y1 = t0 = 0;
  /* write to port to start reading */
  out_byte(JS_PORT, 0x1);
  /* wait for an axis' bit to clear */
  while (--j && (chk = (in_byte(JS_PORT) & js_exist))) {
    if (chk & JS_X_0) {
      t_x0++;
    }
    if (chk & JS_Y_0) {
      t_y0++;
    }
    if ((chk & JS_X_1)) {
      t_x1++;
    }
    if ((chk & JS_Y_1)) {
      t_y1++;
    }
  }
  /* Enable interrupts again */
  unlock();
  /* read values are returned to user */
  js_data[0].js_save.x = t_x0;  js_data[0].js_save.y = t_y0;
  js_data[1].js_save.x = t_x1;  js_data[1].js_save.y = t_y1;
  m_ptr->COUNT = JS_RETURN;
  return(JS_RETURN);
}
```

Programa 2 - Función de Lectura

Para la operación de lectura, en el driver para Linux se dispara un temporizador de la PC (*timer 0*) al inicio del ciclo de lectura. Este temporizador es un contador de 16 bits que una vez arrancado decrementa hasta llegar a cero. Cuando se obtiene un cero en el puerto 0x201h se mira el valor actual de ese contador. La diferencia entre el valor actual del temporizador y aquel al momento de iniciar el ciclo es directamente proporcional a la resistencia en la entrada. Sin embargo, esta solución no se utilizó en el caso de RT-MINIX, sino que se recurrió a simples

contadores en la forma de variables enteras que se incrementaban durante el ciclo. El seudocódigo de la función de lectura se muestra en el Programa 2.

```
struct js_data_type {
     int buttons;
     int x;
     int y;
};
```

Programa 3 - Estrutura empleada por el device driver de joystick

Este cambio se decidió en aras de simplificar el código fuente y para evitar riesgos de interferencia por la reprogramación de los temporizadores cuando se manipula la resolución (o "granularidad"). Para retornar los valores de los ejes x e y junto con el estado de los pulsadores (entradas digitales) se recurre a la estructura de datos del Programa 3.

```
int main ()
{
  int fd, status;
  struct js_data_type js;

  fd = open("/dev/js0", O_RDONLY);         /* open device */
  if (fda < 0) {
    fprintf(stderr, "Device access error\n");
    exit(ERROR);
  }
  while(TRUE) {
    status = read(fd, &js, JS_RETURN);     /* read device */
    if (status != JS_RETURN) {
      fprintf(stderr, "Device reading error\n");
      exit (ERROR);
    }
    fprintf (stdout, "Current readings:  X %4d   Y %4d \r",
      js.x,
      js.y);
  }
  close(fd);                               /* close device */
}
```

Programa 4 - Acceso a la palanca de juegos desde RT-MINIX

MINIX provee los archivos de comando `/usr/bin/MAKEDEV` y `/usr/bin/DESCRIBE`, para crear y describir dispositivos respectivamente; estos archivos se modificaron para contemplar los dispositivos de caracteres `/dev/js0`

y /dev/js1 con número principal 15, número menor 0 y 1. Para usar este nuevo servicio de RT-MINIX, un programa de usuario simplemente utiliza la palanca de juegos como otro dispositivo más. En primer lugar hay que abrir el dispositivo, obteniendo un descriptor de archivo, que se empleará a continuación en las sucesivas lecturas del mismo, hasta que no se necesite más el acceso, momento en que se procede a cerrar el dispositivo. El Programa 4 presenta el cuerpo de un sencillo ejemplo de utilización de la palanca de juegos, ilustrando los pasos descriptos anteriormente.

Colas Unificadas

El planificador original de MINIX consta de tres colas, para poder manejar procesos de categorías tarea, servidor y usuario, en ese orden de prioridad. Cada cola se planifica usando el algoritmo de planificación Round Robin. La Figura 17 muestra la estructura de MINIX en relación a los procesos y el pasaje de mensajes y el manejo y encolado de procesos listos.

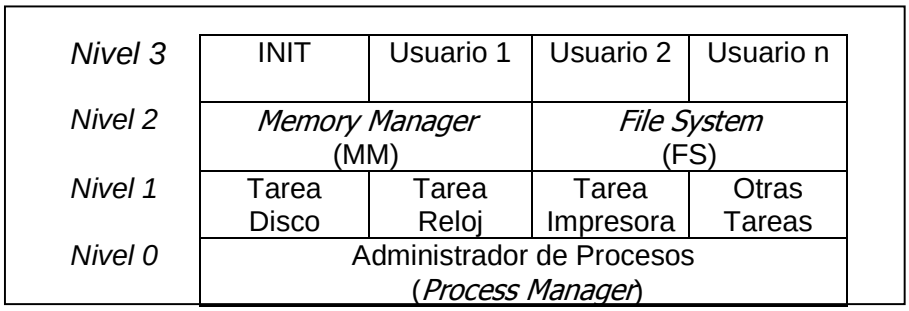

Figura 17 - Estructura de Procesos en MINIX [Tan87]

Cada nivel se describe en el párrafo subsiguiente:

- *Nivel 0* está en cargo de tres tareas fundamentales: manejo de procesos, pasaje de mensajes y manejo de interrupciones.
- *Nivel 1* incluye procesos de E/S o tareas (conocidas también como *device drivers*).
- *Nivel 2* contiene sólo dos procesos, FS y MM, aportando una máquina extendida capaz de manejar llamadas al sistema de cierta complejidad.
- *Nivel 3* comprende todos los procesos debajo del proceso INIT (el primero en ejecutar cuando se inicia el sistema operativo), el lugar para las aplicaciones (como compiladores, editores, comandos) y procesos de usuario.

La idea básica al considerar la posibilidad de unificar las colas en MINIX está relacionada con el objetivo de que una tarea de tiempo real no debería ser interferida por interrupciones de bajo nivel (y sus servidores asociados). [WCG98] trabajó con la hipótesis de que las colas de categoría servidor y usuario pueden ser unidas, permitiendo que los procesos MM y FS sean movidos (disminuyendo

entonces su prioridad) de la categoría proceso servidor a la de proceso de usuario. El resultado esperado de tal cambio es conseguir mejor tiempo de respuesta del sistema operativo. La unión de las colas evita la interferencia de tareas del sistema operativo con la mayoría de las tareas críticas de tiempo real. Teniendo en cuenta varios ejemplos de posibles escenarios, estos casos de estudio y su impacto en los tiempos de procesamiento, resultó claro que la unificación de las colas era factible. Reducir el número de colas es también un paso hacia la tolerancia a fallas.

Con los cambios pensados, hubo que tener muy en cuenta el problema de abrazo mortal (*deadlock*). Cuando la disponibilidad de recursos compartidos (tales como los servicios de MM y FS) disminuye porque ya están siendo usados por distintos procesos, es muy probable que aparezca un abrazo mortal. Esto sucede siempre que un proceso se bloquea esperando por un segundo proceso, mientras que este último también está esperando por el primero. Nuevamente, bajo el planificador original de MINIX 2.0, un proceso que requería un servicio de los procesos MM o FS (asignación de memoria para un arreglo de enteros o acceso a un archivo) lo tenía disponible de inmediato. Esto es así ya que FS o MM tienen la prioridad suficiente para comenzar en cualquier momento sin ser interrumpidos. Se realizó un análisis más en profundidad para comprobar la posibilidad de aparición de situaciones de abrazo mortal entre FS y MM, revisando primero su semántica y luego tratando de medir el impacto del nuevo planificador (aquel con las colas unificadas). La única comunicación posible entre FS y MM (en el código fuente original) se da en la inicialización del sistema, y esa conexión es unidireccional, de este modo evitando el caso de espera circular. Este esquema permite concluir que FS y MM trabajan en forma independiente, teniendo relación sólo con procesos de categoría tarea (el kernel en sí mismo o controladores de dispositivos). Los procesos de nivel tarea tienen la prioridad más alta y no son interrumpidos por esa condición, con su ejecución pudiendo considerarse instantánea (y atómica) en relación a un proceso de usuario.

Ya que Los procesos de usuario no se pueden comunicar entre sí; FS no se comunica con MM; y el manejo de la cola de tareas no fue alterado respecto del código original, se puede concluir que es improbable que suceda abrazo mortal debido a la unificación de colas.

El propósito de esta extensión es mejorar el rendimiento del sistema operativo al reducir la cantidad de colas relacionadas con el manejo de tareas. Esta reducción tiene además ventajas en las características de tolerancia a fallas. El archivo `/usr/src/kernel/proc.c` fue modificado para incorporar la capacidad de manejar colas unificadas o no. La activación o no del modelo de colas unidas se indica por un parámetro de configuración del sistema operativo en el archivo `/usr/include/minix/config.h` (ver el apartado COMPILACIÓN en la sección CÓDIGO FUENTE más adelante).

```
void ready(rp)
register struct proc *rp;  /* this process is now runnable */
{
  ..
  ..
  /* USER_Q queueing can be by Stack or Queue model */
  #if (QUEUEING_MODEL == STACK_TYPE)
    if (rdy_head[USER_Q] == NIL_PROC)
      rdy_tail[USER_Q] = rp;
    rp->p_nextready = rdy_head[USER_Q];
    rdy_head[USER_Q] = rp;
  #else
    if (rdy_head[USER_Q] != NIL_PROC)
      rdy_tail[USER_Q]->p_nextready = rp;
    else
      rdy_head[USER_Q] = rp;
    rdy_tail[USER_Q] = rp;
    rp->p_nextready = NIL_PROC;
  #endif /* QUEUEING_MODEL */
  ..
  ..
}
```

Programa 5 - Manejo de procesos listos en RT-MINIX

Originalmente MINIX provee código para encolar los procesos de usuario según dos métodos: por cola o por pila; sin embargo éste último no se empleaba ya que el código correspondiente estaba comentado en el código fuente original. La

modificación propuesta explicita la forma de manejar la cola de procesos listos mediante el parámetro `QUEUEING_MODEL` en el archivo `/usr/src/kernel/proc.h`, el cual puede tomar dos valores: `STACK_TYPE` o `QUEUE_TYPE`. El Programa 5 muestra un fragmento del código de la función `ready()`, encargada de manejar los procesos listos, con los cambios realizados para insertar ese proceso listo en uno u otro modelo de procesamiento.

Estadísticas de Tiempo Real

Cuando los servicios de tiempo real habían extendido las capacidades de MINIX, se necesitó contar con herramientas de medición, para registrar la evolución de la ejecución de tareas de tiempo real de acuerdo a diferentes estrategias de planificación y a la vez considerar distintas condiciones de carga de trabajo sobre el sistema operativo en general. De esta forma se introdujo una nueva estructura de datos que el sistema operativo mantiene con la información adecuada y que está disponible para el usuario a través de una llamada al sistema.

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

Programa 6 - Estructura de datos para registrar estadísticas de tiempo real

Los elementos que conforman la estructura de datos mostrada en el Programa 6 son:

actpertsk, actapetsk: número de tareas de tiempo real activas (en ejecución), tanto periódicas y aperiódicas.

misperdln, misapedln: número de metas perdidas, tanto periódicas y aperiódicas.

totpertsk, totapetsk: número total de tareas de instancias de tareas de tiempo real activas (en ejecución), tanto periódicas y aperiódicas.

gratio: radio de garantía, esto es, la relación entre número de instancias y metas cumplidas.

idletime: tiempo (en ticks de reloj) no usado como tiempo de cálculo.

```
== RT Statistics ==
                      Periodic -- Aperiodic
Missed Deadlines:       0           0       Guarantee Ratio: 100
Active Tasks:           4           0       Realtime:        8325
Handled Tasks:          7           1       Idletime:        1278
```

Programa 7 - Ejemplo de Estadísticas por pantalla

Los valores estadísticos sobre el rendimiento de RT-MINIX (valores contenidos en la estructura de datos que se está describiendo) pueden consultarse en cualquier momento en pantalla pulsando una tecla de función programada a tal efecto con un formato según se presenta en el Programa 7.

Algoritmos de Sensado Robusto

Propósito

Buscando ampliar las capacidades de tolerancia a fallas de RT-MINIX, el esfuerzo de investigación se dirigió también al campo de los sensores replicados (aprovechando el nuevo servicio de RT-MINIX para el acceso a la palanca de juegos). A partir de [Bro96], los algoritmos utilizados en ese trabajo sirvieron como marco de referencia, y cuatro de ellos se codificaron para ser evaluados.

La primera aproximación fue utilizar directamente los datos provistos en los ejemplos del citado trabajo. En forma estática (con los datos directamente en el código fuente), se logró reproducir el comportamiento de dichos algoritmos frente a ese escenario. Luego de varios refinamientos y cambios a la forma de implementación de los algoritmos, los resultados obtenidos fueron iguales a los presentados por los autores. Luego se pensó en someter dichos algoritmos a un conjunto dinámico (variable en el tiempo) de datos. Para ello se construyó un dispositivo consistente en un conjunto de cuatro potenciómetros, conectados a la entrada de la palanca de juegos (*joystick*) de una PC corriendo RT-MINIX. Cada uno de estos potenciómetros puede pensarse como un sensor (usados en las articulaciones de un brazo robot por ejemplo, para determinar la posición del brazo en el espacio) y sus valores fueron leídos en tiempo real mediante el servicio de acceso a la palanca de juegos de RT-MINIX. El conjunto de valores así obtenidos se sometió al arbitrio de los distintos algoritmos codificados para lograr un valor integrado; valor que pueda luego ser usado por una aplicación de tiempo real para tareas de control. De esta forma también se trabajó con otro de los temas expuestos en el capítulo Estado del Arte: se implementó el concepto de sensor abstracto propuesto en [Mar90] (Sensores Abstractos Tolerantes a Fallas). Con resultados satisfactorios a la vista, el paso siguiente consistió en incorporar esos algoritmos al *kernel* de RT-MINIX y brindar al programador una llamada al sistema para hacer uso de este servicio.

Algoritmos Desarrollados

Los cuatro algoritmos de sensado robusto verificados primero como programas de usuario, y luego incluidos directamente en RT-MINIX se explican a continuación. Se entiende por EP a un elemento de proceso (*PE – Processing Element*), es decir cada sensor empleado. La cantidad de EP (o sensores) es N y se denota con τ a la cantidad de sensores defectuosos. Tener siempre presente que estos algoritmos serán efectivos siempre que (τ < N/2). Para los cálculos, N y τ son parámetros fijados al momento de usar los algoritmos.

Será responsabilidad del programador al emplear este servicio, determinar los valores de N y τ correspondientes. Los algoritmos no tienen por finalidad determinar la cantidad de sensores defectuosos, sino devolver un valor apropiado para un sensor abstracto a partir de un conjunto de lecturas de N sensores, de las cuales τ de ellas se saben (o al menos se intuyen) defectuosas.

Algoritmo:	**Acuerdo Aproximado** (*Approximate-agreement*)
Entrada:	un conjunto de datos S.
Salida:	un nuevo valor, común a todos los anteriores y al cual ellos convergían.
Paso 1:	cada EP anuncia su valor
Paso 2:	cada EP recibe los valores de los otros EPs y los ordena en un vector *V*.
Paso 3:	Los τ menores valores y los τ mayores valores de *V* se descartan.
Paso 4:	Se forma un nuevo vector *V'*, tomando los restantes δ valores donde δ=N-2*τ (los restantes menores valores).
Paso 5:	El nuevo valor es la media de los valores en el nuevo vector *V'* formado en el paso 4.

Algoritmo:	**Convergencia Rápida** (*Fast Convergence*)
Entrada:	un conjunto de EP, cada uno con un valor.
Salida:	un nuevo valor, común a todos los anteriores y al cual ellos convergían.

Paso 1: cada EP recibe los valores de los otros EPs y forma un conjunto *V*.
Paso 2: Los valores aceptables[4] se colocan en un conjunto *A*.
Paso 3: Calcular e(*A*).
Paso 4: Cualquier valor inaceptable en *V* se reemplaza por e(*A*) [5].
Paso 5: El nuevo valor es el promedio de los valores en *V*.

Algoritmo: **Región Optima** (*Optimal Region*)

Entrada: un conjunto de lecturas *S*.
Salida: una región que describe la región que debe ser correcta.

Paso 1: Inicializar una lista de regiones, llamada *C*, a NULL.
Paso 2: Ordenar todos los puntos en *S* en orden ascendente.
Paso 3: Una lectura se considera activa si su límite inferior ha sido cruzado y su límite superior todavía tiene que ser cruzado. Trabajar con la lista en orden, registrando las lecturas activas. Siempre que se alcanza una región donde N - τ o más lecturas están activas, agregar la región a *C*.
Paso 4: Todos los puntos han sido procesados. La lista *C* ahora contiene todas las intersecciones de (N - τ) o más lecturas. Ordenar las intersecciones en *C*.
Paso 5: Devolver la región definida por el menor límite inferior y el mayor límite superior en *C*.

Algoritmo: **Híbrido Brooks-Iyengar** (*Brooks-Iyengar Hybrid*)

Entrada: un conjunto de datos S.
Salida: un valor real dando la respuesta precisa y un rango dando explícitamente los límites de exactitud.

Paso 1: cada EP recibe los valores de todos los otros EPs y forma un conjunto *V*.
Paso 2: Realizar el algoritmo Región Optima sobre *V* y devolver un conjunto *A* que consiste en los rangos de los valores donde al menos N - τ EPs se intersecan.
Paso 3: Determinar el rango definido por el menor límite inferior y el mayor límite superior en *A*. Estos son los límites de exactitud de la respuesta.

[4] Un valor es aceptable si está a una distancia δ de otros N-τ valores

[5] e(*A*) puede ser cualquiera de varias funciones sobre los valores de *A*. Se sugieren promedio, mediana o puntos medios como posibles opciones, que pueden ser apropiadas para diferentes aplicaciones del algoritmo. En nuestro caso se usó promedio.

Paso 4: Sumar los puntos medios de cada rango en *A* multiplicado por el número de sensores cuyas lecturas se intersecan en ese rango y dividirla por el número de factores. Esa es la respuesta.

Pruebas Estáticas

PRUEBA ESTÁTICA INICIAL

Para comprobar que los algoritmos anteriores habían sido correctamente llevados a la práctica, era necesario realizar una serie de pruebas. La primera de ellas consistió en usar datos en forma "estática"; es decir, incluidos directamente en el código fuente del programa escrito para tal fin (ver Programa 8). El conjunto de datos considerado fue tomado de [Bro96] y se presenta en la Tabla 2; simula un conjunto de 5 sensores, uno de los cuales funciona en forma defectuosa, de esta forma emitiendo un valor diferente junto a sus otros cuatro vecinos cada vez que se realiza el proceso de lectura de los mismos.

Lectura	Sensor 1	Sensor 2	Sensor 3	Sensor 4	Sensor 5
Caso 1	4,7 ± 2,0	1,6 ± 1,6	3,0 ± 1,5	1,8 ± 1,0	3,0 ± 1,6
Caso 2	4,7 ± 2,0	1,6 ± 1,6	3,0 ± 1,5	1,8 ± 1,0	1,0 ± 1,6
Caso 3	4,7 ± 2,0	1,6 ± 1,6	3,0 ± 1,5	1,8 ± 1,0	2,5 ± 1,6
Caso 4	4,7 ± 2,0	1,6 ± 1,6	3,0 ± 1,5	1,8 ± 1,0	0,9 ± 1,6

Tabla 2 - Valores considerados en la pruebas estática inicial

Los valores mostrados en la Tabla 2 pueden pensarse, por ejemplo, como el valor en grados del ángulo del codo en un brazo robótico y están expresados en la forma de un valor junto a la tolerancia de la medición (en más y en menos). De este modo se forma un intervalo que representa los posibles valores que la variable física en cuestión (en este caso el ángulo que se está midiendo) podría tomar dados los efectos de muestreo, la incertidumbre del proceso físico y la limitación de precisión propia de cada tipo de sensor empleado. En esta prueba está presente el concepto de *sensor abstracto* según [Mar90]: "un sensor abstracto es un conjunto de valores que contiene la variable física de interés".

Para cada uno de los cuatro casos de la Tabla 2 se aplican sobre los datos de todos los sensores cada uno de los algoritmos disponibles; por lo tanto, en todo momento, el número de sensores (N) es 5 y el número de elementos de proceso defectuosos (τ) es 1. Con estas condiciones se preserva la efectividad de los algoritmos ya que $1 < 5/2$.

```
int main ()
{
  struct sensor_reading readings[NR_PES];
  struct sensor_reading *result;
  int k;

  /* Values for correct PEs */
  readings[0].lbound = 2.7;
  readings[0].value = 4.7;
  readings[0].ubound = 6.7;

  readings[1].lbound = 0.0;
  readings[1].value = 1.6;
  readings[1].ubound = 3.2;
  ..
  ..

  /* Values for malfunctioning PE - case 1 */
  readings[4].lbound = 1.4;
  readings[4].value = 3.0;
  readings[4].ubound = 4.6;

  /* Use every available algorithm */
  printf("Testing algorithms with static data");
  k = rt_robsen(RSA_AAA, readings, result);
  printf("Aproximate Agreement Alg. :  %.2f\n", result->value);
  k = rt_robsen(RSA_ORA, readings, result);
  printf("Optimal Region Alg.       :  [%.2f..%.2f]\n", result-
>lbound, result->ubound);
  k = rt_robsen(RSA_BIH, readings, result);
  printf("Brooks-Iyengar Hybrid Alg.:  [%.2f..%.2f]  %.3f\n",
result->lbound, result->ubound, result->value);
  k = rt_robsen(RSA_FCA, readings, result);
  printf("Fast Convergence Alg.     :  %.2f\n", result->value);
}
```

Programa 8 - Prueba Estática Inicial

Para citar un ejemplo, el conjunto de datos sobre los que se aplicarán los algoritmos dado el caso 1 es: [[2,7; 4,7; 6,7], [0; 1,6; 3,2], [1,5; 3,0; 4,5], [0,8; 1,8; 2,8], [1,4; 3,0; 4,6]] mientras que para el caso 4 los datos a emplear serán: [[2,7; 4,7; 6,7], [0; 1,6; 3,2], [1,5; 3,0; 4,5], [0,8; 1,8; 2,8], [0,7; 1,6; 2,5]]. Se ve claramente como difieren en el quinto elemento, la lectura del sensor defectuoso. Disponer tanto de los datos de entrada (Tabla 2) como de los resultados de salida (ver RESPUESTAS DE CADA ALGORITMO más adelante), fue de gran ayuda en el proceso de validación y corrección de los algoritmos implementados; un proceso de refinamiento gradual hasta que se obtuvieron las respuestas correctas.

SEGUNDA PRUEBA ESTÁTICA

De la bibliografía consultada, surgió la posibilidad de revisar la validez de la implementación propuesta que había sido sometida a la prueba estática inicial, aplicando los algoritmos a otro conjunto de datos. Son los usados en [Jay94] y en ese trabajo se presentan en un gráfico (Figura 18), mostrando a la vez los datos a emplear y los resultados a obtener.

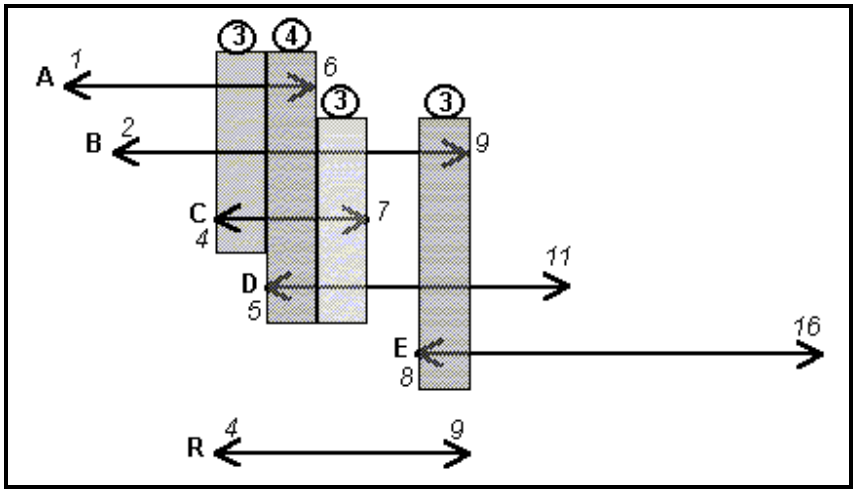

Figura 18 - Rangos y Regiones según [Jay94]

En este caso también se cuenta con cinco sensores, pero esta vez dos de ellos son defectuosos. Con este conjunto de valores de prueba, tampoco peligra la efectividad de los algoritmos pues 2 < 5/2.

Los sensores están representados por las flechas identificadas con las letras desde A hasta E. Nuevamente los valores a emplear están expresados como un rango (para mantener el concepto de *sensor abstracto*, según se definió en la prueba estática inicial), definido por los números en los extremos de cada flecha (para el sensor C el rango correspondiente es [4..7]). Al no estar aclarado en forma explícita en la bibliografía original, se asumió que el valor de cada lectura estaba en el punto medio de cada intervalo. Con este criterio se pudo resumir la información presentada en la Figura 18 mediante la Tabla 3. Los rectángulos sombreados representan las regiones que deben identificar los algoritmos *Región Optima* e *Híbrido Brooks-Iyengar*; y el número sobre los rectángulos indica la cantidad de lecturas que se intersecan en cada una de ellas (valores que emplea el algoritmo *Híbrido Brooks-Iyengar* en su último paso). Con esos datos se verifica la correcta implementación de los algoritmos.

Sensor	***Límite Inferior***	***Valor***	***Límite Superior***
A	1.0	3.5	6.0
B	2.0	5.5	9.0
C	4.0	5.5	7.0
D	5.0	8.0	11.0
E	8.0	12.0	16.0

Tabla 3 - Datos para la Segunda Prueba Estática

Por último, la flecha identificada con la letra R es el rango resultante, y donde deberá encontrarse el valor correcto para el sensor abstracto constituido por esos cinco sensores concretos. Para este caso, se puede aplicar otra vez el ejemplo del conjunto de prueba anterior: los valores representan el ángulo de la posición del codo de un brazo robótico. Como en la prueba estática inicial, se usó el mismo esquema de incluir los valores directamente en el código fuente, según se muestra en el Programa 9.

```
int main ()
{
  struct sensor_reading readings[NR_PES];
  struct sensor_reading *result;
  int k;

  /* Values for PEs */
  readings[0].lbound = 1.0;
  readings[0].value = 3.5;
  readings[0].ubound = 6.0;

  readings[1].lbound = 2.0;
  readings[1].value = 5.5;
  readings[1].ubound = 9.0;
  ..
  ..
  readings[4].lbound = 8.0;
  readings[4].value = 12.0;
  readings[4].ubound = 16.0;

  /* Use every available algorithm */
  printf("Testing algorithms with static data");
  k = rt_robsen(RSA_AAA, readings, result);
  printf("Aproximate Agreement Alg. :  %.2f\n", result->value);
  k = rt_robsen(RSA_ORA, readings, result);
  printf("Optimal Region Alg.       :  [%.2f..%.2f]\n", result-
>lbound, result->ubound);
  k = rt_robsen(RSA_BIH, readings, result);
  printf("Brooks-Iyengar Hybrid Alg.:  [%.2f..%.2f]  %.3f\n",
result->lbound, result->ubound, result->value);
  k = rt_robsen(RSA_FCA, readings, result);
  printf("Fast Convergence Alg.     :  %.2f\n", result->value);
}
```

Programa 9 - Segunda Prueba Estática

Respuestas de Cada Algoritmo

RESULTADOS DE LA PRUEBA ESTÁTICA INICIAL

Para los valores presentados en [Bro96], aplicando los algoritmos implementados se debían obtener los resultados según se describe de aquí en adelante.

Acuerdo Aproximado: como en nuestro escenario τ es 1 (uno sólo de los cinco sensores usados es defectuoso), el algoritmo descarta el menor y mayor valor en cada caso y promedia los valores restantes

C1: Descartar 1,6 y 4,7. Promedio restantes valores es 2,60.
C2: Descartar 1,0 y 4,7. Promedio restantes valores es 2,13.
C3: Descartar 1,6 y 4,7. Promedio restantes valores es 2,43.
C4: Descartar 0,9 y 4,7. Promedio restantes valores es 2,13.

Convergencia Rápida: Todos los valores enviados por el Sensor 5 están dentro de la intersección de al menos otros tres sensores, ninguno debe ser descartado. La respuesta de cada caso es simplemente el promedio de los cinco valores en ese caso.

C1: 2,82.
C2: 2,42.
C3: 2,72.
C4: 2,40.

Región Optima: Se usan los rangos definidos por la incertidumbre en cada caso, incluyendo el sensor defectuoso.

C1: Cuatro EPs se intersecan en [1,5...2,7]. Cinco EPs se intersecan en [2,7...2,8]. Cuatro EPs se intersecan en [1,5...3,2]. La respuesta correcta debe estar en [1,5...3,2].
C2: Cuatro EPs se intersecan en [1,5...2,6]. Cuatro EPs se intersecan en [2,7...2,8]. La respuesta correcta debe estar en [1,5...2,8].
C3: Cuatro EPs se intersecan en [1,5...2,7]. Cinco EPs se intersecan en [2,7...2,8]. Cuatro EPs se intersecan en [2,8...3,2]. La respuesta correcta debe estar en [1,5...3,2].
C4: Cuatro EPs se intersecan en [1,5...2,5]. Cuatro EPs se intersecan en [2,7...2,8]. La respuesta correcta debe estar en [1,5...2,8].

Híbrido Brooks-Iyengar: se realiza un promedio pesado de los puntos medios de los rangos obtenidos por el algoritmo Región Optima (ver resultados del algoritmo anterior).

C1: Promedio pesado: (4 * 2,1 + 5 * 2,75 + 4 * 3,0) / 13 = 2,625.

C2: Promedio pesado: (4 * 2,05 + 4 * 2,75) / 8 = 2,400.

C3: Promedio pesado: (4 * 2,1 + 5 * 2,75 + 4 * 3,0) / 13 = 2,625.

C4: Promedio pesado: (4 * 2,0 + 4 * 2,75) / 8 = 2,375.

Una característica importante a la vista de estos valores es que los algoritmos tienen como respuesta un rango más estrecho que los datos de entrada. En las pruebas realizadas se obtuvieron resultados que comprobaron la implementación, como se muestra en el Programa 10, la salida obtenida para el ejemplo del caso 1.

```
Testing robust sensing algorithms with static data...

Approximate Agreement Alg.:  2.40
Optimal Region Alg.       :  [1.50..3.20]
Brooks-Iyengar Hybrid Alg.:  [1.50..3.20]  2.630
Fast Convergence Alg.     :  2.77
```

Programa 10 - Salida para la prueba estática del caso 1

RESULTADOS DE LA SEGUNDA PRUEBA ESTÁTICA

Para el caso de la segunda prueba estática (usando los datos de [Jay94]), al aplicar los algoritmos implementados se obtuvieron los resultados que muestra el Programa 11, confirmando la implementación de los algoritmos ya que los valores de salida de todos ellos estuvieron dentro del rango de resultado determinado por la flecha con rótulo R (mostrado anteriormente en la Figura 18).

```
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
```

Programa 11 - Salida de la prueba estática con datos según [Jay94]

Prueba Dinámica

Por último, la mejor prueba pensada para los algoritmos era con datos que cambiaran en todo momento. Se construyó para ello un modelo (Figura 19), consistente en cuatro potenciómetros lineales de 100 KΩ conectados a cada una de las cuatro entradas resistivas del puerto de la palanca de juegos de una PC.

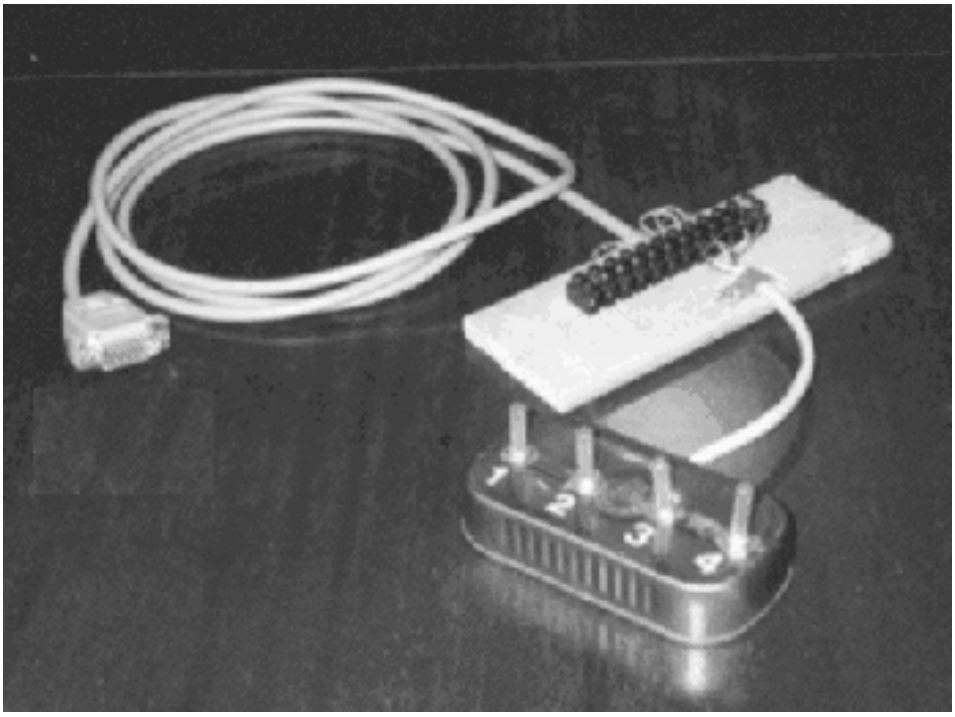

Figura 19 - Dispositivo empleado en la prueba dinámica

El primer paso en la prueba dinámica fue escribir un programa auxiliar para leer continuamente las cuatro entradas al mismo tiempo y presentar los valores en pantalla. Utilizando este programa auxiliar se puede acomodar cada potenciómetro (o "sensor") a un valor cualquiera, ajustando entonces el comportamiento del modelo a diversas situaciones, en particular y de interés para las pruebas, colocar tres potenciómetros con valores iguales y el restante con otro valor totalmente distinto. Así se simula un sensor “defectuoso”, o más precisamente, fuera de rango respecto de los restantes (con este modelo resulta N=4 y τ=1 verificando que $1 < 4/2$, y respetando la efectividad de los algoritmos a probar).

Una vez ajustado el modelo, se corre el programa principal para llevar a cabo la prueba dinámica. La primera tarea es generar un conjunto de lecturas de esos sensores así ajustados. Como los algoritmos esperan un rango en cada lectura de sensor, se realizan tres lecturas consecutivas a una misma entrada y con esos tres valores (ordenados de menor a mayor) se arma una lectura de sensor. Se repite ese proceso para cada uno de los sensores. De esta forma se obtienen cuatro lecturas de sensores (una por cada potenciómetro del modelo); ese conjunto de lecturas se somete entonces a cada uno de los algoritmos codificados, igual a como ocurrió con la prueba estática. La gran diferencia en este caso es la ventaja de poder contar con diferentes conjuntos de valores para realizar toda una serie de corridas (el modelo puede cambiarse de una corrida a otra para probar a los algoritmos con valores distintos cada vez). Para comprender el significado de los números en esta prueba dinámica, el modelo puede pensarse como sensores de posición de una válvula en un conducto, donde el menor valor del potenciómetro corresponde a la válvula totalmente cerrada mientras que el mayor valor corresponde a la válvula totalmente abierta.

```
Testing robust sensing algorithms with dynamic data...

Sensor     L. Bound   Value      U. Bound
  0         14.0       15.0       15.0
  1        683.0      684.0      684.0
  2        683.0      684.0      684.0
  3        683.0      683.0      684.0

Approximate Agreement Alg.:  683.50
Optimal Region Alg.        :  [683.00..684.00]
Brooks-Iyengar Hybrid Alg.:  [683.00..684.00]  683.250
Fast Convergence Alg.      :  516.50
```

Programa 12 - Resultados de la prueba dinámica (corrida A)

Tanto el Programa 12 como el Programa 13 muestran resultados obtenidos en dos de estas corridas cualesquiera. En la pantalla siempre están disponibles los valores usados en ese caso y los resultados al aplicar los algoritmos a ese conjunto de datos.

```
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        313.0      314.0      315.0
  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
```

Programa 13 - Resultados de la prueba dinámica (corrida B)

IMPLEMENTACIÓN

Llevar a la práctica los algoritmos desarrollados en la sección anterior consistió en un proceso gradual. Para tratar las lecturas de un sensor en forma de rango se definió la estructura del Programa 14. Se pensó a una lectura como un valor *v*, con un límite inferior *li* y un límite superior *ls*, determinados éstos por la exactitud del sensor. Se toma entonces un rango, donde el valor leído pertenece al intervalo *[li..ls]*.

```
struct sensor_reading {
     float lbound;
     float value;
     float ubound;
}
```

Programa 14 - Estructura para representar una lectura de sensor

Calcular la región óptima (usado en el algoritmo homónimo y en el Híbrido Brooks-Iyengar) resultó en una aproximación para simplificar el código, ya que C no brinda instrucciones nativas para el manejo de conjuntos. Se decidió trabajar en todo momento con arreglos.

La idea consiste en formar una lista con todos los extremos (inferiores y superiores) de las lecturas (rangos) a considerar y ordenar esa lista en forma ascendente. Quedan así determinados (N*2)-2 regiones (rangos) que son usados cada vez para controlar con los N lecturas cuántas intersecciones hay en cada

región. Aquella región donde haya (N-τ) o más intersecciones se coloca en una lista de regiones a emplear. El proceso se repite hasta que no haya más rangos para controlar. El Programa 15 es el seudocódigo de la solución propuesta.

```
int main()
{
  struct sensor_reading region, regions[NR_PES], values[NR_PES*2];

  /* Prepare a list of regions to be considered */
  for(xi = 0; xi < NR_PES; xi++) {
    values[xi*2].lbound = readings[xi].lbound;
    values[xi*2+1].lbound = readings[xi].ubound;
  }
  /* Sort the values in that list in ascending order */
  ssort(values, NR_PES*2, LBOUND);
  /* Prepare a region and check if intersects all readings */
  for(xi = 0; xi < ((NR_PES*2)-1); xi++) {
    region.lbound = values[xi].lbound;
    region.ubound = values[xi+1].lbound;
    for (yj = 0; yj < NR_PES; yj++) {
      if (intersects(region, readings[yj]))
        region.value++;
    }
    /* Number of intersections in regions OK: add region */
    if (region.value >= (NR_PES - FAULTY_PES))
      nr = addregion(regions, region, nr);
  }
```

Programa 15 - Rutina para encontrar regiones

El dispositivo empleado para proveer de datos variables que fueron usados en la prueba dinámica de los algoritmos de sensado robusto tiene el circuito eléctrico presentado en la Figura 20.

Los algoritmos fueron codificados como funciones separadas incluidas en el archivo `/usr/src/kernel/sensing.c` que son invocadas desde otra función en el archivo `/usr/src/kernel/system.c` encargada de resolver el tipo de algoritmo a aplicar cuando el programador usa la llamada al sistema `rt_robsen()`(este servicio se detalla más adelante en Servicios Disponibles en RT-MINIX).

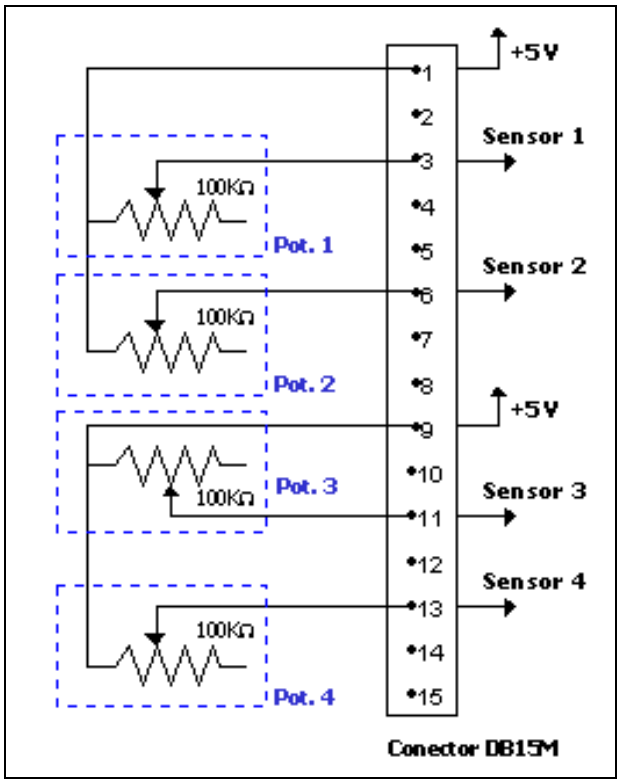

Figura 20 - Diagrama eléctrico para el dispositivo usado en la prueba dinámica

CAMBIOS A LA IMPLEMENTACION

En primera aproximación, la cantidad de sensores a emplear por los algoritmos y la cantidad de sensores defectuosos se prefijaba de antemano en el archivo `/usr/src/kernel/sensing.h` mediante las constantes `NR_PES` y `NR_FAULTY_PES`, respectivamente. Pero para emplear RT-MINIX con una cantidad distinta de sensores, el programador estaba obligado a recompilar el núcleo (*kernel*) del sistema operativo luego de cambiar el valor de dichas constantes. Esta práctica no es buena, ya que no permite una reconfiguración rápida de las aplicaciones del usuario, con la obligación de compilar el kernel ante cada cambio en el número de sensores a emplear. Para subsanar esta situación se decidió usar la estructura de datos mostrada en el Programa 16, que sirve para indicar el tipo de algoritmo deseado junto a la cantidad sensores (N) y número de sensores defectuosos (τ). La llamada al sistema entonces valida que no se comprometa la efectividad de los algoritmos (calculando que se cumpla $\tau < N/2$) y

en caso afirmativo, invoca la función apropiada, devolviendo así al programador el resultado de la lectura del sensor abstracto.

```
struct rsa_params{
      int type;
      int nr_pes;
      int faulty_pes;
}
```

Programa 16 - Estructura para utilizar algoritmos de sensado robusto

Este cambio implicó asignar a los arreglos una cantidad fija de elementos, lo suficientemente grande como para satisfacer las necesidades más comunes de sensores a utilizar, pero sin incurrir en un desperdicio de memoria en el kernel. Ya que una lectura es una estructura de 12 bytes, reservar 16 sensores como valor máximo, no genera un impacto significativo en el tamaño final de la imagen del kernel y permite trabajar en las aplicaciones del usuario con un número variable de sensores sin necesidad de tener que compilar el kernel ante cada cambio en la cantidad de sensores a usar.

RESUMEN DE LAS CORRIDAS

Corrida	***Sensores***											
	1			***2***			***3***			***4***		
	li	***v***	***ls***	***li***	***v***	***ls***	***li***	***v***	***ls***	***li***	***v***	***ls***
A	14,0	15,0	15,0	683,0	684,0	684,0	683,0	684,0	684,0	683,0	683,0	684,0
B	578,0	638,0	677,0	614,0	626,0	688,0	313,0	314,0	315,0	604,0	649,0	681,0
C	473,0	480,0	507,0	480,0	504,0	538,0	307,0	308,0	308,0	478,0	492,0	550,0
D	492,0	503,0	507,0	433,0	506,0	517,0	308,0	308,0	308,0	482,0	529,0	531,0
E	506,0	516,0	546,0	488,0	517,0	520,0	307,0	307,0	308,0	480,0	510,0	513,0
F	535,0	610,0	703,0	519,0	616,0	660,0	148,0	218,0	308,0	525,0	609,0	680,0
G	675,0	686,0	709,0	658,0	682,0	682,0	148,0	148,0	148,0	674,0	688,0	689,0
H	565,0	638,0	688,0	592,0	658,0	706,0	148,0	179,0	202,0	609,0	647,0	705,0
I	555,0	631,0	702,0	532,0	656,0	759,0	167,0	193,0	201,0	553,0	610,0	657,0
J	651,0	667,0	685,0	625,0	706,0	730,0	191,0	192,0	192,0	647,0	668,0	686,0
K	678,0	679,0	680,0	679,0	680,0	681,0	193,0	193,0	194,0	672,0	679,0	680,0
L	678,0	679,0	682,0	681,0	681,0	681,0	194,0	194,0	194,0	677,0	679,0	681,0

Tabla 4 - Valores usados en las corridas de la prueba dinámica

Según se mencionó más arriba, disponer del modelo permitió realizar distintas corridas con condiciones diversas de los sensores. La Tabla 4 presenta el

resumen de todas las corridas llevadas a cabo, mostrando para cada sensor la lectura correspondiente, en la forma de tres columnas: un límite inferior (li), un valor (v) y un límite superior (ls). Los resultados obtenidos al aplicar cada algoritmo disponible en RT-MINIX a las lecturas de cada corrida se detallan en la Tabla 5; están expresados según el tipo de algoritmo, por un valor (v), un rango con límite inferior (li) y límite superior (ls) o un rango y un valor. Los algoritmos están identificados por sus iniciales: AA, Acuerdo Aproximado; RO, Región Optima; HBI, Híbrido Brooks-Iyengar y CR, Convergencia Rápida.

Corrida	***Resultados Algoritmos***						
	AA	***RO***		***HBI***			***CR***
	v	***li***	***ls***	***li***	***ls***	***v***	***v***
A	683,50	683,00	684,00	683,00	684,00	683,25	516,50
B	632,00	614,00	677,00	614,00	677,00	645,50	556,75
C	486,00	480,00	507,00	480,00	507,00	493,50	446,00
D	504,50	492,00	507,00	492,00	507,00	499,50	461,50
E	513,00	506,00	513,00	506,00	513,00	509,50	462,50
F	609,50	535,00	660,00	535,00	660,00	597,50	475,00
G	682,00	675,00	682,00	675,00	682,00	678,50	551,00
H	642,50	609,00	688,00	609,00	688,00	648,50	530,50
I	620,50	555,00	657,00	555,00	657,00	606,00	522,50
J	667,50	651,00	685,00	651,00	685,00	668,00	558,25
K	679,00	679,00	680,00	679,00	680,00	679,50	557,75
L	679,00	678,00	681,00	678,00	681,00	679,50	558,25

Tabla 5 - Resultados obtenidos para las corridas de la prueba dinámica

Comparación de los Algoritmos

En base a la experiencia adquirida tanto en la implementación (pasar del pseudo-código a código C corriendo bajo RT-MINIX) como con las pruebas realizadas (ya sea estáticas como dinámicas) se puede realizar una comparación de los algoritmos en distintas áreas: por construcción, por velocidad de respuesta y por resultados, y que se amplía a continuación.

POR CONSTRUCCIÓN

La codificación de los algoritmos no presentó grandes problemas a partir del pseudo-código disponible ni diferencias sustanciales entre cada uno de ellos.

POR VELOCIDAD DE RESPUESTA

No se apreciaron diferencias en la velocidad de respuesta en cualquiera de los cuatro algoritmos.

POR RESULTADOS

Una primera consideración es que el tipo de respuesta es distinto según el algoritmo empleado. Tanto *Convergencia Rápida* como *Acuerdo Aproximado* expresan el resultado como un valor, *Región Optima* con un rango mientras que *Híbrido Brooks-Iyengar* con un rango y un valor. Desde este punto de vista, *Híbrido Brooks-Iyengar* se presenta como el más completo: al emplearlo se dispone del valor para el sensor abstracto y además de un rango que indica la exactitud de esa respuesta.

Si se considera ahora el valor de los resultados, los algoritmos *Región Optima* e *Híbrido Brooks-Iyengar* tienen como respuesta un rango más estrecho que los datos de entrada, algo que se aprecia claramente en ambas corridas de la prueba dinámica (Programa 12 y Programa 13). El hecho de que los rangos sean idénticos en los dos algoritmos se debe a que usan la misma rutina para encontrar regiones (Programa 15). Hay que notar también que el valor devuelto por *Acuerdo Aproximado* siempre se encuentra dentro del rango encontrado por los algoritmos *Región Optima* e *Híbrido Brooks-Iyengar*. Hasta aquí no habría entonces diferencias en emplear cualquiera de los algoritmos. Sin embargo, se aprecia que entre el resultado de *Acuerdo Aproximado* y el valor devuelto por *Híbrido Brooks-Iyengar* hay una diferencia que es mayor cuanto mayor es la amplitud del rango resultado: en la corrida A (Programa 12) la amplitud es 684-683=1 y la diferencia es 683,5-683,25=0,25; en la corrida B (Programa 13) la amplitud es 677-614=63 y la diferencia es 645,5-632=13,5.

Para comprobar si esta deducción se podía generalizar, se calculó la amplitud del rango respuesta (tomando *ls* - *li*) del algoritmo *Híbrido Brooks-Iyengar* (HBI) y la

diferencia (en valor absoluto) entre el valor de BHI y el del algoritmo *Acuerdo Aproximado* (AA), para todas las corridas.

Corrida	*Amp. Rango HBI (2)*	*Dif.Valor AA y HBI (1)*	*Relación (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%

Tabla 6 - Análisis de Resultados entre Algoritmos

También se obtuvo la relación porcentual entre estos dos datos. Esa información está contenida en la Tabla 6, la cual se presenta ordenada en forma ascendente por la segunda columna (rango resultado).

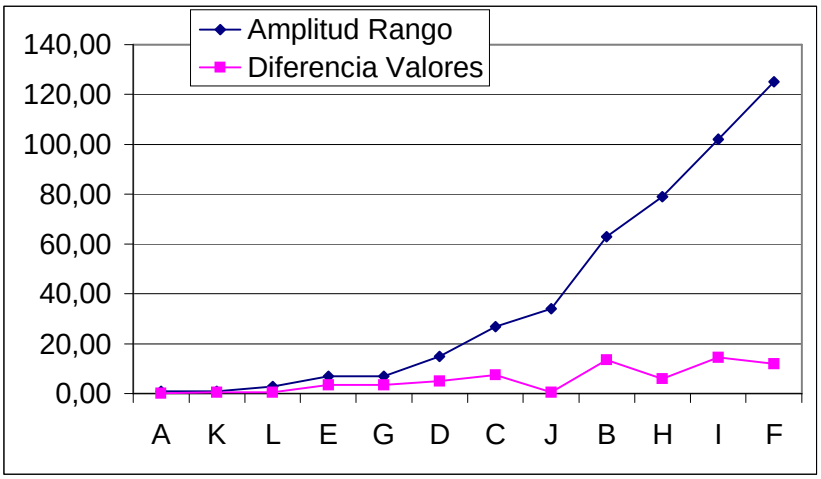

Figura 21 - Comparación de Resultados entre Algortimos

A partir de la Tabla 6 se realizó un gráfico (Figura 21), donde se puede apreciar la tendencia, para confirmar la deducción realizada: a mayor amplitud en el rango resultado de HBI, mayor será es la diferencia entre el valor resultado de dicho algoritmo y aquel del algoritmo AA. Se puede inferir entonces que es conveniente usar el algoritmo *Híbrido Brooks-Iyengar* en el caso de emplear sensores con un intervalo de precisión amplio.

Servicios Disponibles en RT-MINIX

El conjunto de extensiones de tiempo real realizadas en RT-MINIX queda encapsulado al programador de aplicaciones en una biblioteca de tiempo real que contiene todas las llamadas que éste puede efectuar al sistema operativo para hacer uso de los distintos servicios. Todas esas llamadas se encuentran en el archivo `/usr/src/lib/rtlib.c` y para usarlas el programador debe incluir siempre en su aplicación el archivo `/usr/include/minix/rt.h` que incorpora todos los prototipos y definiciones necesarias para acceder a los servicios de tiempo real.

Llamada	***Descripción***	***Categoría***
`rt_pertsk`	Crea una tarea periódica	Manejo de Tareas
`rt_apetsk`	Crea una tarea aperiódica	
`rt_tskend`	Termina una tarea periódica	
`rt_killper`	Termina todas las tareas periódicas	
`rt_setgrain`	Establece el "grano"[6] del reloj del sistema	Manejo del Reloj
`rt_getgrain`	Devuelve el "grano" del reloj del sistema	
`rt_uptime`	Devuelve el tiempo[7] desde que se inició el sistema	
`rt_getstats`	Devuelve las estadísticas[8] de tiempo real	Estadísticas
`rt_resetstats`	Pone a cero las estadísticas de tiempo real	
`rt_robsen`	Aplica un algoritmo de sensado robusto a un conjunto de lecturas de sensores	Algoritmos de Sensado Robusto

Tabla 7 - Llamadas de tiempo real en RT-MINIX

La Tabla 7 resume dichas llamadas, agrupadas por categoría de servicios y a continuación se describe la funcionalidad de cada una de las llamadas de tiempo real:

[6] Grano: cantidad de ticks por segundo

[7] En ticks

[8] Según la estructura `rt_globstats`

RT_PERTSK

Esta llamada al sistema crea una tarea periódica de tiempo real. El prototipo de la llamada es el siguiente:

```
int rt_pertsk(int Ti, int Ci, int priority);
```

En el prototipo anterior:

- *Ti* es un entero positivo que indica el período (en "ticks") de la tarea.
- *Ci* es un entero positivo usado para determinar el peor tiempo de ejecución de la tarea.
- *priority* es la prioridad que el usuario asigna a la tarea.

RT_APETSK

Esta llamada al sistema crea una tarea aperiódica de tiempo real. El prototipo de la llamada es el siguiente:

```
int rt_apetsk(int Ti, int Ci, int priority);
```

En el prototipo anterior:

- *Ti* es un entero positivo que indica el tiempo en que se iniciará la tarea.
- *Ci* es un entero positivo usado para determinar el peor tiempo de ejecución de la tarea.
- *priority* es la prioridad que el usuario asigna a la tarea.

RT_TSKEND

Esta llamada al sistema termina inmediatamente la tarea periódica de tiempo real que se está ejecutando. El prototipo de la llamada es el siguiente:

```
int rt_tskend();
```

RT_KILLPER

Esta llamada al sistema termina inmediatamente todas las tareas periódicas de tiempo real que están planificadas para ser ejecutadas. El prototipo de la llamada es el siguiente:

```
int rt_killper();
```

RT_GETGRAIN

Esta llamada al sistema devuelve la resolución (o "grano") actual del reloj del sistema. El prototipo de la llamada es el siguiente:

```
long rt_getgrain();
```

RT_SETGRAIN

Esta llamada al sistema establece una nueva resolución (o "grano") del reloj del sistema. El prototipo de la llamada es el siguiente:

```
void rt_setgrain(long newgrain);
```

En el prototipo anterior:

- *newgrain* es el nuevo valor a ajustar en el reloj del sistema. Indica la cantidad de veces por segundo que el sistema será interrumpido para realizar la planificación.

RT_UPTIME

Esta llamada al sistema devuelve el tiempo desde que se inició el sistema operativo. El prototipo de la llamada es el siguiente:

```
clock_t rt_uptime();
```

RT_RESETSTATS

Esta llamada al sistema vuelve a cero todas las estadísticas de tiempo real que el sistema operativo mantiene. El prototipo de la llamada es el siguiente:

```
int rt_resetstats();
```

RT_GETSTATS

Esta llamada al sistema devuelve las estadísticas de tiempo real que el sistema operativo mantiene. El prototipo de la llamada es el siguiente:

```
int rt_getstats(struct rt_globstats *st);
```

En el prototipo anterior:

- *st* es un puntero a una estructura de datos en el área de memoria del usuario donde el sistema operativo copiará los valores estadísticos al momento de la llamada.

RT_ROBSEN

Esta llamada al sistema emplea alguno de los cuatro algoritmos de sensado robusto para determinar cual es el valor confiable dado un conjunto de lecturas de un grupo de sensores. El prototipo de la llamada es el siguiente:

```
int rt_robsen(struct rsa_params params, struct
     sensor_reading readings[], struct
     sensor_reading result[]);
```

En el prototipo anterior:

- *params* es una estructura de datos que indica el tipo de algoritmo a utilizar [alguno de estos valores: AAA (*Aproximate Agreement Algorithm*), FCA (*Fast Convergence Algorithm*), BIH (*Brooks-Iyengar Hibrid*) y ORA (*Optimal Region Algorithm*)], la cantidad de sensores empleados (< MAX_PES) y la cantidad de sensores defectuosos (< , datos necesarios para determinar el valor apropiado del sensor abstracto.
- *readings* es un arreglo de las lecturas de los distintos sensores.
- *result* es la lectura correcta luego de aplicar el algoritmo pedido al conjunto de lecturas efectuadas.

Tecla de Función	***Tarea***
F6	Poner a cero cantidad metas perdidas
F8	Mostrar en pantalla las estadísticas de tiempo real
F10	Cambiar el algoritmo de planificación (RMS↔EDF)

Tabla 8 - Acciones de teclas de función en RT-MINIX

Adicionalmente se dispone de ciertos servicios que pueden ejecutarse “sobre la marcha”, empleando teclas de función reprogramadas a tal efecto, según se detalla en la Tabla 8.

Resumen

MINIX 2.0 se convirtió en RT-MINIX, un sistema operativo con extensiones de tiempo real. Servicios tales como creación y manejo de tareas de tiempo real (periódicas o aperiódicas), planificación de tiempo real (usando los algoritmos RMS y EDF), un nuevo *device driver* para los conversores AD incorporados en la PC y algoritmos de sensado robusto hacen de esta plataforma una opción más que interesante a la hora de elegir una plataforma para desarrollar sistemas de tiempo real. MINIX cubrió con creces las expectativas respecto de la capacidad para el desarrollo de un sistema operativo de tiempo real. La disponibilidad del código fuente completo, junto a la pequeña cantidad de archivos fuentes que lo componen, permitió entender en forma clara y modificar los componentes que fueran necesarios. Así todos los cambios y agregados realizados se integraron directamente en el *kernel* del sistema operativo, y se proveyó una biblioteca de soporte.

5. Detalles de Implementación

"Trabajo pesado es por lo general la acumulación de tareas livianas que no se hicieron a tiempo"

H. Cooke

Introducción

Presentadas en la sección anterior todas las extensiones de tiempo real que incorpora RT-MINIX, corresponde aquí desarrollarlas desde un punto de vista detallado en cuanto a los alcances de los cambios o agregados que fueron necesarios realizar a una distribución MINIX 2.0 original. Las extensiones realizadas se tratan por separado. Para cada tema se describen los archivos modificados o nuevos, además de las estructuras de datos involucradas y se discuten aspectos puntuales explicando las soluciones usadas, comparándolas en algunos casos con otras alternativas disponibles y citando el criterio a la hora de su elección.

La idea de esta sección es presentar una guía práctica para entender los alcances de las experiencias con el modelo real, y brindar material de referencia y consulta para todos aquellos que quieran extender RT-MINIX más allá de las características provistas en la presente versión. Se presentan también indicaciones para obtener, instalar y compilar RT-MINIX.

Descripción de las Extensiones

Manejo de Tareas de Tiempo Real

Un sistema de tiempo real consta de una serie de tareas que deben llevarse a cabo en un momento determinado. Para asegurarse que la restricciones temporales o metas de cada tarea puedan cumplirse, se emplean distintos algoritmos de planificación. Para implementar el concepto de tarea de tiempo real en RT-MINIX, se modificó la estructura propia de un proceso, incorporando las

propiedades adicionales como se muestra en el Programa 17 y que se realizaron en el archivo `/usr/src/kernel/proc.h`.

```
struct proc {
  ..
  ..
#if ENABLE_REALTIME
  int task_type;      /* RT: type of task: periodic or aperiodic */
  int priority;       /* RT: task priority. Unused */
  int Ti;             /* RT: period - deadline if aperiodic task */
  int Ci;                  /* RT: worst execution time */
  clock_t next_period;     /* RT: start time of next period */
  int misdln;              /* RT: missed deadlines of task */
#endif /* ENABLE_REALTIME */
};
```

Programa 17 - Propiedades de una tarea de tiempo real

Cuando se crea una tarea con la llamada `rt_pertsk()` o `rt_apetsk()`, las funciones en realidad cambian las características del proceso que se había creado, para lo cual se modificó el archivo `/usr/src/kernel/proc.c`. Primero se asigna el tipo de tarea, junto a los valores de período si es periódica o meta si es aperiódica y el peor tiempo de ejecución de la tarea. Se ha reservado una propiedad para indicar la prioridad de la tarea que se está creando. Por el momento esta propiedad no se usa (para todas las tareas este campo tiene el mismo valor), pero en futuras versiones de RT-MINIX los algoritmos de planificación con prioridades podrán emplearla directamente. Además se actualizan las estadísticas de tiempo real respecto de la cantidad de tareas totales y activas en cada caso.

RESOLUCIÓN DEL RELOJ

MINIX trata al reloj como una tarea de E/S más, ya que considera que atiende a un dispositivo: el reloj físico de la PC y por lo tanto, presenta la estructura mencionada anteriormente. Se introdujeron cambios a los archivos `/usr/src/kernel/clock.c` y `/usr/src/kernel/clock.h` que posibilitan variar la resolución o "grano" del reloj en RT-MINIX. Se proveen dos llamadas en la biblioteca de tiempo real tanto para cambiar como para obtener el grano actual del reloj. Este se expresa en "ticks", es decir, cantidad de interrupciones por

segundo. Al iniciarse RT-MINIX conserva la resolución normal de MINIX, es decir de 60 ticks (60 Hz, la frecuencia de la red eléctrica). Queda a cargo del programador cambiarla para ajustarla a sus necesidades.

Hay que notar que a mayor resolución del reloj, mayor es el impacto en el rendimiento, pues la tarea del reloj se ejecutará mayor cantidad de veces en el mismo lapso de tiempo. Las pruebas realizadas en [Wai95] demostraron que para valores de hasta 10000 ticks por segundo, el comportamiento del sistema no estaba afectado por esta recarga, permitiendo así manejar una precisión de 100 microsegundos. Para comprobar que esto seguía siendo válido en la nueva versión de RT-MINIX (esta vez empleando una PC Pentium 100 MHz con 32 MB de memoria), se realizó una sencilla prueba consistente primero en cambiar la granularidad del sistema (con la llamada `rt_setgrain()`) y luego lanzando una tarea periódica de tiempo real cualquiera (con la llamada `rt_pertsk()`). Estos pasos se repitieron para más de treinta casos, cambiando cada vez el valor de la resolución (o "grano") del reloj en forma creciente. Nuevamente no se encontró recarga para valores hasta 10000 ticks por segundo, mientras el sistema se mostró inestable cuando se superaba esa marca (evidenciado por sucesivas "colgadas" en cada caso).

ESTADÍSTICAS DE TIEMPO REAL

Las estadísticas de tiempo real que se llevan durante el funcionamiento del sistema operativo están accesibles al programador como vimos por medio de una llamada al sistema. Se describen a continuación las acciones necesarias para mantener cada uno de esos valores:

actpertsk: se incrementa en uno al crear una tarea periódica. Se reduce en 1 al terminar una tarea periódica.

actapetsk: se incrementa en uno al crear una tarea aperiódica. Se reduce en 1 al terminar la tarea aperiódica.

totpertsk, totapetsk: se incrementa en 1 cada vez que se crea una tarea periódica o aperiódica, respectivamente.

misperdln: se incrementa en 1 cada vez que una tarea periódica pierde su meta, es decir, el tiempo actual es mayor que el próximo período de la tarea.

misapedln: se incrementa en 1 si la tarea aperiódica no termina antes del tiempo requerido como meta.

gratio: es el cociente entre la suma de metas perdidas (periódicas y aperiódicas) dividida la suma de las tareas totales (periódicas y aperiódicas) multiplicado por 100.

idletime: se le asigna el valor del campo user_time de ese proceso (actualizado continuamente por la tarea del reloj).

A todas los variables anteriores se les asigna el valor 0 (excepto *gratio* que recibe 100) en la función `do_reteststat()` que se encarga de reiniciar las estadísticas del sistema.

Los archivos `/usr/src/kernel/system.c`, `/usr/src/kernel/proc.c`, `/usr/src/kernel/clock.c` fueron actualizados para llevar a cabo esa serie de cálculos. Para que los resultados puedan visualizarse en pantalla se han cambiado los archivos `/usr/src/kernel/keyboard.c` y `/usr/src/kernel/dmp.c`.

Biblioteca de Llamadas al Sistema

Los servicios de tiempo real mencionados anteriormente en Servicios Disponibles en RT-MINIX están implementados como llamadas al sistema operativo; es decir, el código fuente de las rutinas que proveen esos servicios forma parte de los archivos que componen el núcleo (*kernel*) del SO. Como cualquier otra llamada al sistema, debe escribirse además la interfaz con las aplicaciones del usuario, código que se incorpora en una biblioteca de soporte.

PASAJE DE MENSAJES

Como muestra la Figura 22, cuando un programa de usuario hace uso de algún servicio de tiempo real (una llamada al sistema), se genera un pasaje de mensajes entre los distintos niveles del SO hasta llegar al lugar donde realmente se realizará la acción pedida. Un programa de usuario invoca el servicio de tiempo real rt_xxxx(), con todos los parámetros necesarios (paso 1). Esa función, contenida en la biblioteca de tiempo real (en la práctica forma parte de la biblioteca general de soporte, ya provista por MINIX y modificada ahora para RT-MINIX), realiza las acciones necesarias a la precondiciones de ese servicio, y si todo está correcto, envía un mensaje al SO (paso 2).

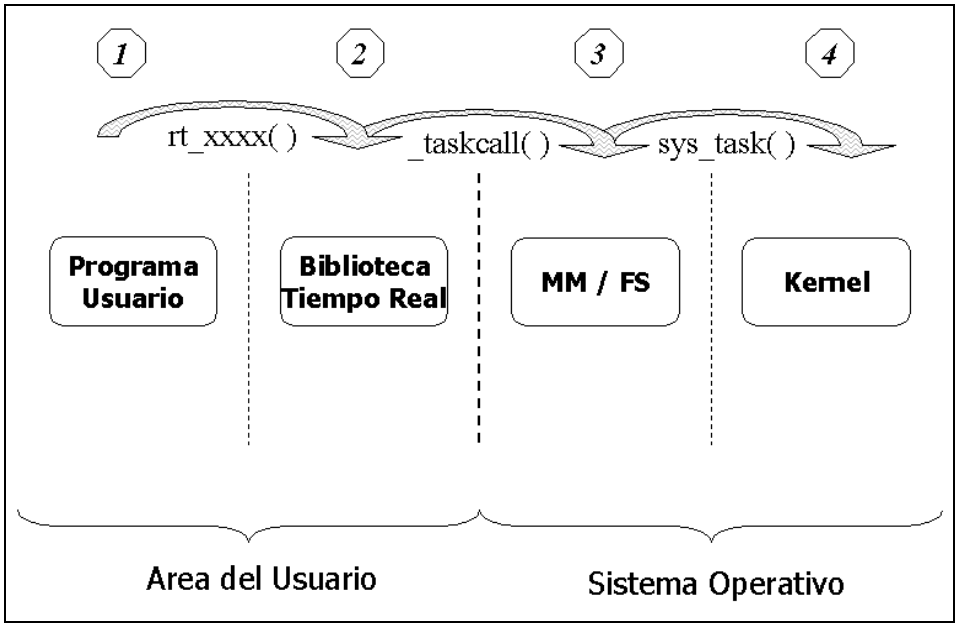

Figura 22 - Secuencia de mensajes para un servicio de tiempo real

A nivel del administrador de memoria (MM) o sistema de archivos (FS), hay una tabla que sirve para relacionar los números de llamadas al sistema en las rutinas que las llevarán a cabo. Una vez decodificado el número de llamada, la función correspondiente envía un nuevo mensaje, esta vez, dirigido al núcleo del SO (paso 3). Este último mensaje también es decodificado y se invoca la función apropiada, donde en definitiva se realiza la tarea deseada (crear una nueva tarea de tiempo

real, ajustar la resolución del reloj, obtener las estadísticas de tiempo real, etc.) (paso 4). Terminada la tarea (brindado el servicio), se completa el camino de mensajes en sentido inverso, con el propósito de hacer llegar al programa de usuario el resultado de su pedido: éxito o error. En todos los casos, hay controles en cada paso para asegurarse el estado de la operación.

IMPLEMENTACIÓN

Para cada uno de los servicios de tiempo real disponible hay una función de la forma que muestra el Programa 18. La función recibe los parámetros apropiados, verifica las precondiciones, envía el mensaje al SO indicando: nro. de llamada que necesita, quién la proveerá (MM o FS) pasando los parámetros por intermedio de un puntero. Luego controla el estado de esa llamada y verifica las postcondiciones, para devolver al usuario el código de estado resultante.

```
public rt_xxxx {
parameter A;
parameter B;
...
if (preconditions() == OK)
      status = _taskcall(who, syscallnr, msgptr);
      /* who = MM or FS
         syscallnr: system call to perform
         msgprt: pointer to message with parameters */
      status = status & postconditions();
else
      status = ERROR;
return(status);
};
```

Programa 18 - Cuerpo de un servicio de TR en la biblioteca de soporte

Todas esas funciones se codificaron en un único archivo para facilitar el mantenimiento. Originalmente en MINIX la biblioteca de soporte está organizada como una serie de subdirectorios bajo `/usr/src/lib`. Se mantuvo esta estructura creando un nuevo subdirectorio `/usr/src/lib/rt` donde se ubicaron los archivos `rtlib.c` y `rtlib.h`. También se modificó el archivo `/usr/src/lib/Makefile` para facilitar la compilación de la biblioteca. Todas las constantes para identificar las distintas llamadas están en `/usr/include/minix/com.h` y se modificaron los

archivos `/usr/src/fs/table.c` y `/usr/src/mm/table.c` que contienen las tablas que relacionan los números de llamadas con las funciones que las realizan.

CÓMO AGREGAR UN NUEVO SERVICIO

En caso de necesitar incorporar una nueva llamada, se debe agregar una nueva función en el archivo `/usr/src/lib/rt/rtlib.c`, su prototipo en `/usr/include/minix/rt.h`, definir el valor de la nueva llamada en `/usr/include/minix/com.h` y modificar la tabla en `/usr/src/mm/table.c`.

Código Fuente

Disponible en http://www.dc.uba.ar/people/proyinv/cso/rt-minix se encuentran las instrucciones para descargar e instalar un paquete con el código fuente necesario para convertir una instalación MINIX 2.0.0 en RT-MINIX (que incluye todos los servicios y cambios descriptos anteriormente). También se puede descargar otro paquete con un conjunto de programas para pruebas de aplicaciones de tiempo real que corren sobre RT-MINIX.

Instalación

1. Copiar el archivo *RT-MINIX.TAR.Z* en el directorio `/usr/src`
2. Ejecutar *uncompress rt-minix.tar.Z*
3. Se obtiene el archivo `rt-minix.tar`
4. Ejecutar *tar xvf rt-minix.tar* que copiará todos los archivos requeridos a sus directorios respectivos (ver listado de archivos nuevos/modificados más abajo).
5. Configurar y compilar el nuevo sistema operativo (se describe a continuación).

Compilación

Se llevó a cabo una modificación en los procedimientos a la hora de compilar un nuevo kernel del sistema operativo. Aplicando una de las técnicas de tolerancia a fallas para evitar errores de programación, se ha cambiado el archivo `/usr/src/kernel/Makefile` para que ahora tenga en cuenta la configuración que se haga del sistema operativo a partir del archivo `/usr/include/minix/config.h`. En este último es posible indicar que servicios se desean incluir cambiando la clave correspondiente (0 para excluir el servicio o 1 para incluirlo). Por ejemplo, para habilitar las funciones de sonido se pone en 1 el valor del parámetro `ENABLE_SB_AUDIO` (ver Tabla 9 para una lista de todos los posibles parámetros). Antes del cambio propuesto, en el kernel se compilaban archivos objetos (*.o) innecesarios, que no guardaban relación con la configuración pedida.

En el momento de compilar un nuevo kernel se ejecuta el comando `make hdboot` en el directorio `/usr/src/tools`. Se emplea el archivo `Makefile` disponible allí, que a su vez compila cada uno de los componentes necesarios. Este procedimiento cambia a los directorios apropiados (`/usr/src/kernel`, `/usr/src/fs`, `/usr/src/mm` y si fuese necesario `/usr/src/inet`) para utilizar luego los archivos `Makefile` que encuentra en cada uno de dichos directorios.

El archivo `/usr/src/kernel/Makefile` fue cambiado, agregando una sección config (que pasa a ser la opción por omisión) donde a partir de un archivo de comandos ya provisto en MINIX (`/usr/src/tools/tell_config`) es posible saber el valor de cada uno de los parámetros de configuración (ver Tabla 9).

Parámetro	**Descripción**	**Objetos**[9]
`ENABLE_JOYSTICK`	Habilita la tarea para el controlador de dispositivos de la palanca de juegos	joystick.o
`ENABLE_SB_AUDIO`	Habilita la tarea para el controlador de dispositivos de placas de sonido SoundBlaster	sb16_dsp.o sb16_mixer.o
`ENABLE_MITSUMI_CDROM`	Habilita la tarea para el controlador de dispositivos para CD-ROM Mitsumi	mcd.o
`ENABLE_ADAPTEC_SCSI`	Habilita la tarea para el controlador de dispositivos para placas SCSI Adaptec	aha_scsi.o
`ENABLE_AT_WINI`	Habilita la tarea para el controlador de dispositivos del disco rígido (versión AT)	at_wini.o
`ENABLE_BIOS_WINI`	Habilita la tarea para el controlador de dispositivos del disco rígido (versión BIOS)	bios_wini.o

[9] Estos archivos serán incluidos o excluidos de la imagen al compilar en el directorio `/usr/src/kernel`

Parámetro	Descripción	Objetos[9]
ENABLE_XT_WINI	Habilita la tarea para el controlador de dispositivos del disco rígido (versión XT)	xt_wini.o
ENABLE_ESDI_WINI	Habilita la tarea para el controlador de dispositivos del disco rígido (versión ESDI)	esdi_wini.o
ENABLE_JOINQUEUE	Habilita el código para unificar las colas de tareas FS y MM	código en varios archivos
ENABLE_REALTIME	Habilita el código para extensiones de tiempo real	código en varios archivos
ENABLE_NETWORKING	Habilita el código TCP/IP	ne2000.o wdeth.o dp8390.o

Tabla 9 - Parámetros de configuración del kernel en RT-MINIX

En base a ello, se construye un archivo temporal con el nombre de los archivos objetos a incluir para luego al final de este proceso, invocar a un nuevo archivo de comandos (`/usr/src/kernel/Makekrnl`) que se encarga de ejecutar `make kernel` con una variable OPTOBJS que contiene los nombres los objetos en el archivo temporal, que luego es borrado.

De esta forma se compilan en el kernel sólo aquellos módulos que correspondan a la configuración indicada. Se logra así reducir al mínimo el tamaño de la imagen resultante, ocupando en memoria el menor espacio posible y limitando la posibilidad de errores de ejecución por defectos de programación en el código fuente.

Archivos nuevos y modificados

A continuación se listan todos los archivos que han sido modificados o creados para incorporar las extensiones de tiempo real a MINIX. Todas las ubicaciones son relativas al directorio `/usr`. Aquellos marcados con * son archivos nuevos.

```
bin/DESCRIBE
bin/MAKEDEV
include/errno.h
include/signal.h
include/time.h
include/minix/callnr.h
```

```
include/minix/com.h
include/minix/config.h
include/minix/const.h
include/minix/joystick.h          *
include/minix/rt.h                *
src/rt-dist                       *
src/fs/Makefile
src/fs/misc.c
src/fs/proto.h
src/fs/table.c
src/fs/time.c
src/kernel/clock.c
src/kernel/console.c
src/kernel/const.h
src/kernel/dmp.c
src/kernel/glo.h
src/kernel/joystick.c             *
src/kernel/kernel.h
src/kernel/keyboard.c
src/kernel/Makefile
src/kernel/Makekrnl
src/kernel/proc.c
src/kernel/proc.h
src/kernel/proto.h
src/kernel/sensing.c              *
src/kernel/sensing.h              *
src/kernel/system.c
src/kernel/table.c
src/lib/Makefile
src/lib/rt/Makefile               *
src/lib/rt/rtlib.c                *
src/lib/rt/rtlib.h                *
src/mm/forkexit.c
src/mm/getset.c
src/mm/main.c
src/mm/Makefile
src/mm/param.h
src/mm/proto.h
```

```
src/mm/signal.c
src/mm/table.c
src/mm/utility.c
```

Aplicaciones de Ejemplo

Para probar las capacidades aportadas por las nuevas extensiones de tiempo real disponibles en RT-MINIX, se escribieron una serie de programas de ejemplo que se ubicaron en los directorios `/usr/local/rt/sensing` y `/usr/local/rt/samples` y cuyo propósito se describe en la Tabla 10.

Programas	***Descripción***
/usr/local/rt/samples	
`aper.c`	Genera una tarea aperiódica
`clock.c`	Muestra el tiempo desde el arranque del sistema operativo tanto en "ticks" como en segundos
`grain.c`	Muestra y cambia el "grano" (resolución) del reloj del sistema (en "ticks" por segundo).
`pertasks.sh`	Lanza varias tareas periódicas para probar la planificación
`sensors.c`	Usa los cuatro ejes de los joysticks como sensores abstractos
`stats.c`	Emplea la llamada al sistema para obtener las estadísticas de tiempo real y las muestra en pantalla
`tempcntl.c`	Controla periódicamente un valor de "temperatura" (lanzando una tarea periódica y usando el eje X del joystick A como sensor)
/usr/local/rt/sensing	
`adjust.c`	Lee continuamente las entradas resistivas de ambos joysticks y muestra las lecturas en pantalla para permitir ajustar el valor de cada una.
`dyntest.c`	Aplica los algoritmos de sensado robusto a un conjunto de lecturas dinámicas obtenidas de "sensores" conectados al conversor A/D de la palanca de juegos. Muestra los resultados por pantalla.
`statest.c`	Aplica los algoritmos de sensado robusto a un conjunto de lecturas estáticas, incluidas directamente en el código fuente. Muestra los resultados por pantalla.

Tabla 10 - Programas de Ejemplo

6. Conclusiones

"Lo bueno, si breve, dos veces bueno"

Gracián

Logros obtenidos

El propósito de este trabajo fue presentar el estado del arte en el campo de la tolerancia a fallas en sistemas de tiempo real y la validación de ciertos conceptos mediante el uso de un modelo real. Se ha mostrado a lo largo de este documento como dos áreas que cierto tiempo atrás discurrían por caminos totalmente separados, se han integrado de manera completa, pudiendo considerarlas en estos momentos como un área de investigación en sí misma.
La tolerancia a fallas es una cualidad que un sistema debe tener para ser confiable, y los sistemas de tiempo real, por su naturaleza (al estar condicionado por el tiempo de sus respuestas) y ámbito de aplicación, deben ser confiables. Dada la creciente complejidad de los nuevos sistemas de tiempo real que se están desarrollando, como las actividades en las cuales serán empleados, donde las fallas pueden resultar catastróficas, se impone que la tolerancia a fallas deba ser una cualidad intrínseca en ellos.

De acuerdo a los temas abarcados a lo largo del presente trabajo, y junto a las experiencias obtenidas con la interacción con un modelo real, se está en condiciones de presentar los siguientes comentarios al respecto:

- MINIX 2.0 sirvió como plataforma para obtener un sistema operativo con extensiones de tiempo real, basado en [Wai95] y tal como se extendió en [Rog99].

- Se dispone de un paquete distribuible de RT-MINIX, adecuado como plataforma para futuros proyectos de tiempo real. Aún así, los servicios

disponibles hasta el momento pueden ser complementados con otra serie de servicios y funciones (los cuales se detallan en la subsección Trabajo Futuro, a continuación).

- Comprobación de la validez y exactitud de algoritmos de sensado robusto, mediante valores estáticos y dinámicos; estos últimos provistos directamente por el SO por un grupo de “sensores” replicados leídos en tiempo real, de acuerdo a las experiencias con un modelo real.

- Se cuenta con herramientas para que el ciclo completo de desarrollo e implementación de un sistema de tiempo real incluya capacidades de tolerancia a fallas:

    - Lenguajes para especificación y diseño con extensiones de tolerancia a fallas (TLA, LaRC, Estructura de Transformaciones) posibilitan la oportunidad de incluir esas características desde la concepción de los sistemas.

    - La planificación de tareas, muestra la diversidad de algoritmos que incluyen capacidad de tolerancia a fallas, ya sea a partir de la variación de algoritmos existentes o probados, o mediante ideas noveles.

    - La profusión de sensores y actuadores, sobre todo en el desarrollo de sistemas autónomos (como aquellos empleados en la exploración espacial) hace de la idea de redundancia un concepto clave para la tolerancia a fallas

- Capacidad para investigar la capacidad de trabajar sensores replicados o entornos multisensores, donde la integración de información o fusión de sensores es crítica para el tratamiento de los datos obtenidos por esos elementos.

- Conocimiento de otros proyectos de sistemas de tiempo real (tal el caso de KURT [Sri98] o RT-Linux [Yod95]). La comparación con estos trabajos permitió comprobar que el esquema empleado en RT-MINIX es acertado, ya que modifica directamente el *kernel* (cambiando primero la capacidad del SO para trabajar con distintos "granos" de reloj, al igual que se hace en KURT con el módulo UTIME incorporados al *kernel* de Linux). Se diferencia bien de la forma de trabajo exhibida en RT-Linux (donde se desarrolló una capa ejecutiva de tiempo real mínima y sobre ella se ha montado Linux, como una tarea de tiempo real más); esquema que trae aparejados problemas en el acceso a los dispositivos (vídeo, disco, salidas serie y paralelo).

Tareas por realizar

Durante el desarrollo de la presente tesis, ya sea durante la etapa de investigación sobre el estado del arte, como luego en el período de experimentación con el modelo real, surgió la necesidad de acotar las áreas de alcance del trabajo. Al mismo tiempo, a medida que se comprobaban ciertas características o capacidades también hubo que limitar las tareas a llevar a cabo, con el propósito de lograr objetivos tangibles. A partir de esas condiciones, se anotaron posibles trabajos a realizar en un futuro, a saber:

- Mantener las distintas actualizaciones y distribuciones de RT-MINIX que se desarrollen, con la ayuda de alguna herramienta para control y manejo de versiones (tal como CVS/RCS).

- Agregar y completar variedad de algoritmos de planificación en RT-MINIX, especialmente aquellos que contemplen la capacidad de tolerancia a fallas en sus cálculos.

- Extensión de RT-MINIX con la capacidad de manejar tareas con prioridades. Eso implicará cambios en los algoritmos de planificación para

que tengan en cuenta dicha característica a la hora de planificar y eviten los casos de inversión de prioridades.

- Probar la factibilidad de incorporar una interface gráfica (X Window por ejemplo) en RT-MINIX, para facilitar el uso de las aplicaciones del usuario, donde la comunicación con el operador es más que importante. A través de distintos controles u objetos se podrá mejorar la indicación de alarmas (en forma visual y auditiva) o presentar los datos registrados o archivos de registro (*logs*) en diversas ventanas al mismo tiempo.

- Someter a RT-MINIX a trabajos de mayor carga (conexión en red con otras máquinas, etc.) y que puedan necesitar más de una computadora/procesador. Con el propósito de probar a RT-MINIX con una gran cantidad de tares de tiempo real, se planteó la necesidad de desarrollar un modelo apropiado. Para ello se pensó en implementar el modelo propuesto en [War85] que representa una línea embotelladora. Las tareas y sus restricciones son ejemplos claros para que un SOTR provea al desarrollador de un conjunto de servicios adecuados para lograr llevar a cabo dichos requisitos.

- Proponer nuevos algoritmos de sensado robusto donde se contemplen condiciones distintas a las limitaciones impuestas a los usados en este trabajo (trabajo con sensores multidimensionales por ejemplo).

7. Bibliografía

"No gasto mi inteligencia en algo que puedo encontrar en un libro"

A. Einstein

Referencias

[And81] T. Anderson, P. Lee, “Fault Tolerance: Principles and Practice”, Prentice Hall, Londres, 1981.

[And82] T. Anderson, P. Lee, “Fault tolerance terminology proposals”, Proc. 12th Int. Symp. on Fault-Tolerant Computing, Los Angeles, Junio 1982, pp. 29-33.

[Avi85] A. Avizienis, “The N-Version Approach to Fault-Tolerant Software”, IEEE Trans. on Software Engineering, SE-11(12):1491-1501, Diciembre 1985.

[Bro96] R. Brooks, S. Iyengar, “Robust Distributed Computing and Sensing Algorithm”, IEEE Computer, Junio 1996, pp. 53-60.

[Che78] L. Chen, A. Avizienis, “N-version programming: A fault tolerant approach to reliability of software operation”, *Dig. 8th Annual Int. Conf. Fault Tolerant Comput.* (FTCS-8), Francia, Junio 1978, pp. 3-9.

[DiV90] B. DiVito, R. Butler, J. Caldwell, “Formal design and verification of reliable computing platform for real-time control”. NASA Tech. Memo 102716, NASA Langley Research Center, Octubre 1990.

[Fer93] C. Ferrell, “Robust Agent Control of an Autonomous Robot with Many Sensors and Actuators”, AITR-1443, Artificial Intelligence Laboratory, MIT, Mayo 1993.

[Gho96] S. Ghosh. “Guaranteeing Fault Tolerance through Scheduling in Real-Time Systems”. PhD Thesis, University of Pittsburgh, Agosto 1996.

[Gho97] S. Ghosh, R. Melhem, D. Mossé, “Fault-Tolerant Rate-Monotonic Scheduling”, Proc. of 6th IFIP Conference on Dependable Computing for Critical Applications (DCCA-97), Marzo 5-7, 1997.

[Jay94] D. Jayasimha, “Fault Tolerance in a Multisensor Environment”, Department of Computer Science, The Ohio University, Mayo 1994.

[Kim94] H. Kim, K. Shin. “Evaluation of Fault-Tolerance Latency from Real-Time Application’s Perspectives”. Technical Report CSE-TR-201-94. CSE Division, EECS Department, The University of Michigan, 1994.

[Lam85] L. Lamport, P. Melliar-Smith, “Syncronizing clocks in the presence of faults”. Journal of the ACM, 32(1):52-78, Enero 1985.

[Lam90] L. Lamport, “A temporal logic of actions”. Reporte Técnico, Digital SRC, California, Abril 1990.

[Lap85] J. Laprie, “Dependable Computing and Fault Tolerance: Concepts and Terminology”, 15th Annual International Symposium on Fault-Tolerant Computing, Junio 1985, pp. 2-11.

[Lap90] J. Laprie, “Definition and Analysis of Hardware and Software Fault Tolerant Architectures”, IEEE Computer, 23(7):39-51, Julio 1990.

[Liu73] C. Liu, J. Layland, "Scheduling Algorithm for Multiprogramming in a Hard Real-Time Environment", Journal of the ACM, 20(1):46-61, Enero 1973.

[LSP82] L. Lamport, R. Shostak , M. Pease, “The Byzantine Generals Problem”, ACM Trans. on Programming Languages and Systems, 4(3):382-401, 1982.

[Mar90] K. Marzullo, “Tolerating failures of continuous-valued sensors”, ACM Transactions on Computer Systems, 8(4):284-304, Nov. 1990.

[Pau98] V. Paulik, Joystick device driver for Linux, disponible en línea en ftp://atrey.karlin.mff.cuni.cz/pub/linux/joystick/joystick-0.8.0.tar.gz

[Pay92] D. Payton, "Do Whatever Works: A Robust Approach to Fault-Tolerant Autonomous Control", Journal of Applied Intelligence, 2:225-250.

[Rog99] P. Rogina, G. Wainer, “New Real-Time Extensions to the MINIX operating system”, Proc. of 5th Int. Conference on Information Systems Analysis and Synthesis (ISAS'99), Agosto 1999.

[Rus91] J. Rushby, "Formal Specification and Verification of a Fault-Masking and Transient-Recovery Model for Digital Flight-Control Systems", Technical Report CSL-91-3, SRI International, Junio 1991.

[Smi88] J. Smith, “A Survey of Software Fault Tolerance Techniques”, Computer Science Department, Columbia University, 1988.

[Sri98] B. Srinivasan. KURT: The KU Real-Time Linux. Disponible en línea en http://hegel.itc.ukans.edu/projects/kurt/

[Sta92a] J. Stankovic, “Real Time Computing”, BYTE, invited paper, pp. 155-160, Agosto 1995.

[Sta92b] J. Stankovic, “The Integration of Scheduling and Fault Tolerance in Real-Time Systems”, *COINS Technical Report 92-49*, Marzo1992.

[Sta96] J. Stankovic et al., “Real-Time Computing: A Critical Enabling Technology”, Umass Technical Report, Julio 1994.

[Tan87] A. Tanenbaum, “A Unix clone with source code for operating systems courses”, *ACM Operating Systems Review*, 21:1, Enero 1987.

[Wai95] G. Wainer, “Implementing Real-Time Scheduling in a Time-Sharing Operating System”, *ACM Operating Systems Review*, Julio 1995.

[War85] P. Ward, S. Mellor, “Structured Development for Real-Time Systems”, Apéndice B, Yourdon Press, 1985.

[WCG98] N. Wolowick, M. Cuenca Acuña, G. Wainer, “Uniendo las colas de planificación en el sistema operativo MINIX”, Reporte Interno, Depto Computación, FCEyN, UBA, Julio 1998.

[Yod95] V. Yodaiken, “Cheap operating systems research and teaching with Linux”, Dept. Of Computer Science, New Mexico Tech., 1995.

[Zhi96] L. Zhiming, J. Mathai, “Verification of Fault-Tolerance and Real-Time”, Dept. of Computer Science, RR302, University of Warwick, Junio 1996.

Glosario

Durante la investigación y preparación de este trabajo, los términos detallados a continuación se han usado con los siguientes significados:

Término Original	***Traducción***
Containment	Represión
Deadlock	Abrazo Mortal
Dependability	Confiablilidad
Device Driver	Controlador de Dispositivos
Failure	Avería
Fault	Falla
Joystick	Palanca de Juegos
Paper	Artículo
Polled	Encuestado
Predictability	Predicción
Preemption	Remoción
Reliability	Seguridad
Resiliency	Elasticidad
Scheduling	Planificación
Slack	Período de poca actividad