6

SISTEMAS TOLERANTES A FALLOS

6.1. Conceptos generales sobre tolerancia a fallos

En este capítulo estudiaremos las técnicas para conseguir que los sistemas continúen funcionando correctamente, a pesar de fallos en su hardware o errores de software. Este tipo de sistemas se denominan **sistemas tolerantes a fallos**.

Los conceptos relacionados con la tolerancia a fallos tienen cada vez más importancia; esto se debe a la proliferación de los sistemas de cómputo y el uso de éstos cada vez en más ámbitos. Algunas de las aplicaciones de los computadores resultan lo suficientemente críticas como para protegerlas contra potenciales fallos. En esos entornos, un funcionamiento incorrecto del sistema resultaría catastrófico y podría causar importantes perjuicios.

Ejemplos de aplicaciones de este tipo son las centrales nucleares controladas por computador, y por extensión, cualquier otro proceso industrial delicado controlado de esa forma. Otros ejemplos son los computadores que gobiernan aviones, satélites artificiales y naves espaciales. Un tipo de aplicaciones que deben ser especialmente tolerantes a fallos, son aquéllas en las que se prevea un funcionamiento continuo durante largo tiempo sin posibilidad de intervención (léase, por ejemplo, naves espaciales de largo recorrido que pueden estar varios años en el espacio, especialmente las no tripuladas).

Definiremos ahora tres conceptos fundamentales en el ámbito de la tolerancia a fallos:

- Llamaremos **fallo** a cualquier defecto, físico o lógico, en cualquier componente, hardware o software, de un sistema. Dentro de esta categoría incluiremos los contactos accidentales entre conductores eléctricos, cortes en los mismos, defectos en los componentes, variaciones en el funcionamiento de los elementos electrónicos debidas a perturbaciones externas (tales como la temperatura u ondas electromagnéticas), etc. Diremos que un fallo se enmarca en el **universo físico**.

- Un **error** es la manifestación o el resultado de un fallo. Dicho de otra forma, un error es la consecuencia de un fallo desde el punto de vista de la información. Los errores se enmarcan dentro del llamado **universo informacional**.

Departamento de Informática
Universidad de Valladolid

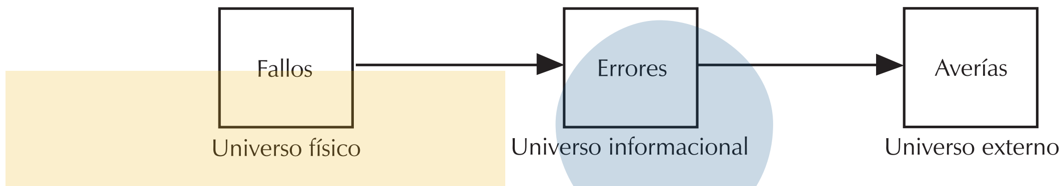


Fig. 6.1. Fallos, errores y averías.

- Si un error causa un funcionamiento incorrecto del sistema desde el punto de vista externo, es decir, si las consecuencias del fallo trascienden al exterior del sistema, diremos que se ha producido una **avería**. Las averías se producen en el **universo externo** o **universo del usuario**.

En la figura 6.1 se muestra la relación entre fallos, errores y averías.

Para aclarar estos conceptos, tomaremos como ejemplo un circuito combinacional cualquiera. Si un punto de ese circuito se conecta incorrectamente, de forma transitoria o permanente, al valor lógico 0, estaremos ante un fallo. Si este hecho lo miramos desde el punto de vista de la tabla de verdad del circuito, nos encontraremos con un error, ya que dicha tabla habrá cambiado como consecuencia del fallo. Cuando ese error afecte a otros circuitos, externos al anterior, o a la salida del sistema, tendremos una avería

En este punto podemos establecer dos nuevas definiciones:

- La **latencia de un fallo** es el tiempo que transcurre desde que se produce un fallo hasta que se manifiesta el error.
- La **latencia de un error** es el tiempo transcurrido entre la aparición de un error y la manifestación de ese error en el exterior del sistema.

6.2. Causas de los fallos

Para buscar todas las causas de los fallos deberemos analizar todo el proceso seguido desde el diseño del sistema hasta su explotación, pasando por su implementación. En todas estas fases puede estar el origen de los fallos.

En primer lugar, puede ya haber fallos causados por errores en el diseño, tanto del hardware como del software, estos errores se concretan en **especificaciones erróneas**. Siguiendo el proceso, otra fuente de fallos la encontraremos en la implementación que definiremos como el proceso de transformar las especificaciones, tanto hardware como software, en sendos entes reales. En este proceso también puede haber errores (**errores de implementación**) debidos a no seguir correctamente las especificaciones, no codificar correctamente el software, seleccionar los componentes incorrectamente, etc. Otra causa de problemas son los **fallos en los componentes**, el origen último de este tipo de fallos es muchas veces difícil de determinar: en algunas ocasiones, los fallos en los componentes pueden deberse a errores en el dimensionamiento de los mismos, que provocan su sobrecarga y su posterior destrucción; otras veces es la corrosión o el desgaste natural el que provoca fallos en los componentes; por otra parte, hay componentes

OCW Departamento de Informática Universidad de Valladolid

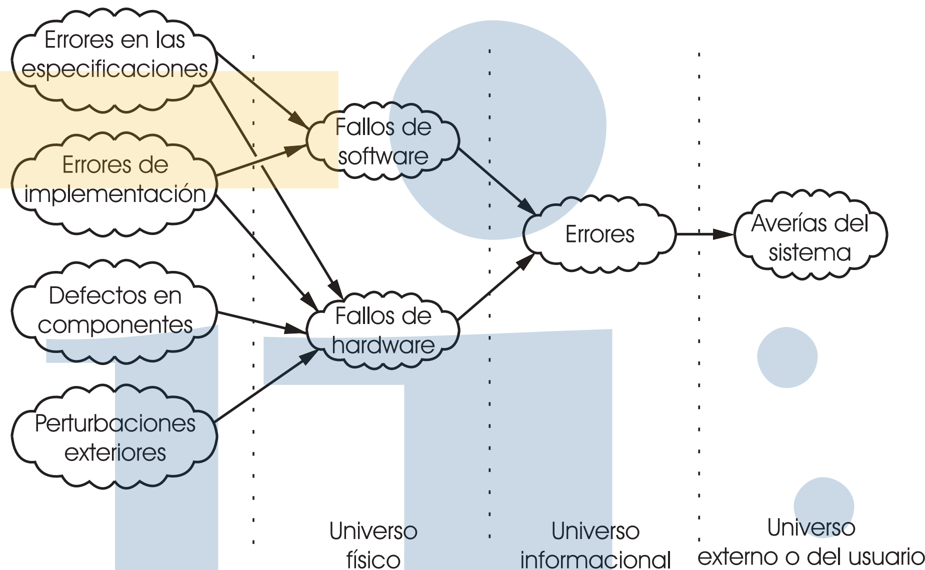


Fig. 6.2. Posibles causas de los fallos.

defectuosos en origen, pero cuyo defecto no se manifiesta hasta pasado cierto tiempo de funcionamiento (recuérdese el concepto de latencia de un fallo mencionado en el apartado anterior). Por último, una de las causas que provocan gran número de fallos son las **perturbaciones externas**, dentro de este concepto se engloban las variaciones de las condiciones ambientales tales como presión, temperatura, humedad, etc., las perturbaciones debidas a campos electromagnéticos e, incluso, los errores del operador (que en un buen diseño deben estar previstos). No hay que olvidar, dentro de las perturbaciones exteriores, los daños producidos en algunos sistemas militares por el fragor de la batalla, accidentes, etc.

En la figura 6.2 se muestran esquemáticamente las causas de los fallos y sus consecuencias en los universos mencionados con anterioridad.

6.3. Caracterización de los fallos

Los fallos pueden caracterizarse atendiendo a varios criterios: causa, naturaleza, duración, extensión y variabilidad (Nelson & Carrol, 1982):

- Las **causas** de los fallos puede ser múltiples, como se vio en la sección anterior: especificaciones incorrectas en el momento del diseño, fallos en el proceso de implementación, defectos en los componentes, perturbaciones externas, etc.
- La **naturaleza** de los fallos especifica la parte del sistema que falla: software o hardware, dentro del hardware, el fallo puede tener naturaleza analógica o digital.
- En cuanto a la **duración**, los fallos pueden ser **permanentes**, que se caracterizan por continuar indefinidamente en el tiempo si no se toma alguna acción correctora; **intermitentes**, que aparecen, desaparecen y pueden reaparecer, de forma repetida y aleatoria, y

Departamento de Informática Universidad de Valladolid

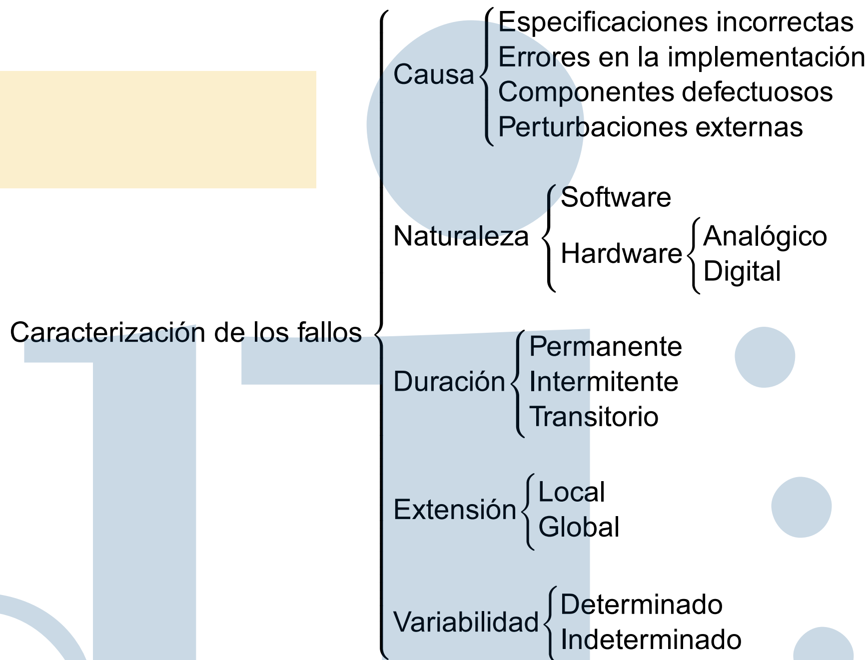


Fig. 6.3. Caracterización de los fallos

transitorios, que aparecen únicamente durante breves instantes coincidiendo con alguna circunstancia, tal como puede ser el encendido o alguna perturbación externa.

- La **extensión** de un fallo indica si sólo afecta a un punto localizado o si afecta a la globalidad del hardware, del software o de ambos.
- Por ultimo, en cuanto a la **variabilidad**, los fallos pueden ser **determinados**, si su estado no cambia con el tiempo, incluso aunque cambie la entrada u otras condiciones, o **indeterminados**, cuyo estado puede cambiar cuando varíen algunas de las condiciones.

Todas estas características de los fallos pueden verse en el esquema mostrado en la figura 6.3.

6.4. Filosofías de diseño para combatir los fallos

Existen tres técnicas básicas de mantener el funcionamiento correcto de un sistema digital ante posibles fallos del mismo. Estas técnicas básicas son las siguientes (figura 6.4):

Prevención de fallos es cualquier técnica capaz de evitar los fallos antes de que se produzcan. Estas técnicas incluyen la revisión de los diseños, análisis minucioso de los componentes empleados, pruebas, controles de calidad, etc. Puede también incluirse aquí el aislamiento del sistema ante perturbaciones externas tales como ruidos electromagnéticos o variaciones de temperatura, presión, etc.

Departamento de Informática Universidad de Valladolid

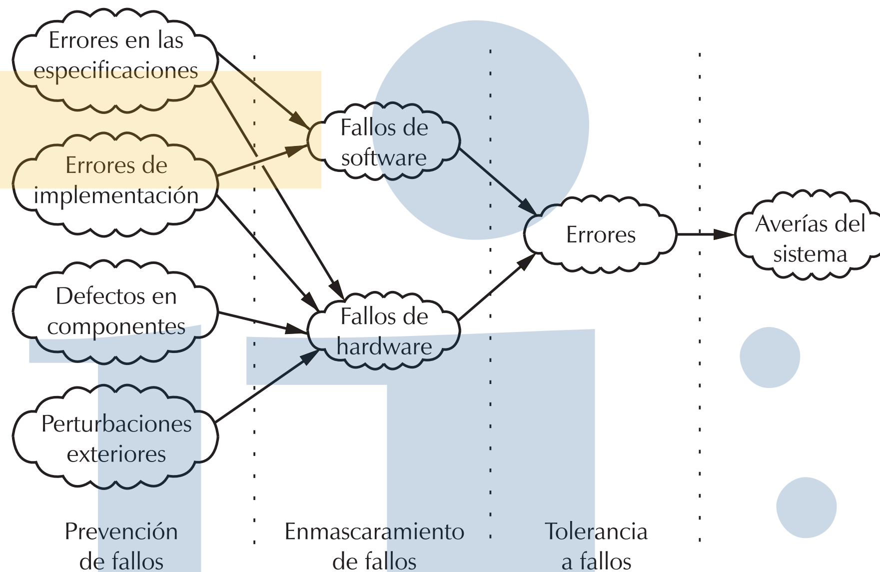


Fig. 6.4. Diferentes clases de barreras para evitar la propagación de los fallos.

Enmascaramiento de los fallos: dentro de esta categoría se encuentran las técnicas que encubren las consecuencias de un fallo. Como ejemplo de estas técnicas podemos poner la utilización de bits redundantes en una memoria con el fin de corregir posibles errores en la memoria, de esta forma corregimos el dato erróneo antes de que el sistema lo use. Mediante estos métodos no evitamos los fallos, pero sí sus consecuencias.

Tolerancia a fallos: ésta puede definirse como la capacidad de un sistema para continuar funcionando normalmente después de producirse un fallo. En general, cuando ocurre un fallo es necesaria la **reconfiguración** del sistema, es decir, la puesta en funcionamiento de elementos redundantes para sustituir al componente causante del fallo.

6.5. Redundancia

Tradicionalmente se ha asociado el concepto de redundancia con la duplicación de los elementos físicos que componen un sistema. Ésta es una de las técnicas más frecuentemente usadas para combatir los fallos. Sin embargo, debe entenderse el concepto de redundancia de una forma más general.

Definiremos la redundancia como *el empleo de información, recursos o tiempo adicionales, por encima de los estrictamente necesarios para el correcto funcionamiento de un sistema*. La redundancia puede adquirir una de las siguientes formas (Johnson, 1984):

Redundancia hardware o física: consiste en incorporar hardware adicional, normalmente con el fin de detectar fallos o conseguir la tolerancia ante los mismos.

Redundancia software: ésta consiste en añadir en los programas líneas de código adicionales para evitar errores. Este código suplementario puede tener diferentes funciones, tales

Departamento de Informática Universidad de Valladolid

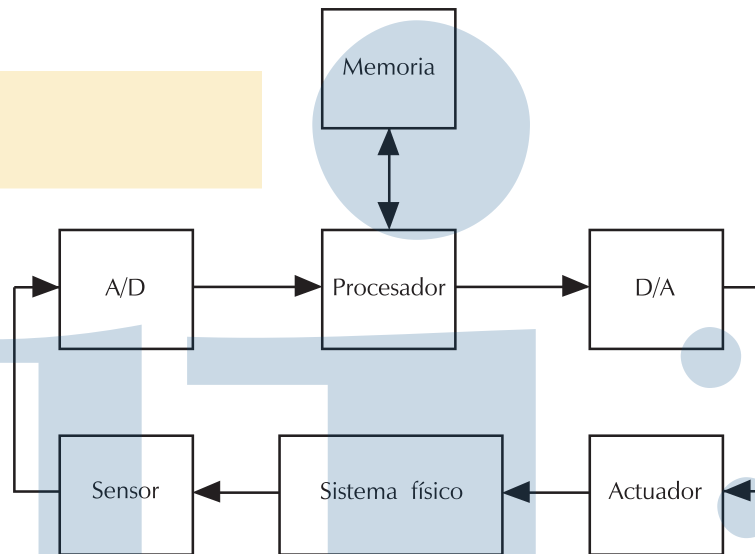


Fig. 6.5. Ejemplo de sistema de control por computador.

como evitar que los datos se salgan de rango, prevenir errores aritméticos, etc.

Redundancia informacional: este tipo de redundancia implica el manejo de información suplementaria con el fin de detectar o corregir errores potenciales.

Redundancia temporal: consiste en emplear tiempo adicional de proceso para detectar posibles fallos, o en su caso, corregirlos. Esto puede suponer, por ejemplo, la repetición de cálculos por diferentes métodos, etc.

Estos tipos de redundancia no son excluyentes y, en muchas ocasiones, varios de ellos funcionan juntos: por ejemplo, la redundancia software implica redundancia temporal, salvo si se emplea un procesador suplementario para ejecutar las instrucciones adicionales, en cuyo caso existirá redundancia hardware; por otra parte, la redundancia informacional puede hacer necesario un incremento de memoria para almacenar la información adicional (redundancia hardware).

Para explicar las diferencias entre los distintos tipos de redundancia, tomaremos como ejemplo un sistema de control computerizado como el mostrado en la figura 6.5: el procesador toma muestras de cierta magnitud del sistema físico, a través del sensor y el convertidor analógico/digital, realiza ciertos cálculos y elabora una respuesta para llevar la magnitud al valor deseado; esta respuesta será enviada a través del convertidor digital/analógico y el actuador. Si ese sistema sólo tiene los medios, tanto hardware como software, estrictamente necesarios para realizar las funciones anteriores, diremos que no posee redundancia. Supongamos ahora que, en ese mismo sistema, se mejora el software de forma que puedan detectarse, por ejemplo, valores incorrectos en el sensor (mediante comprobaciones del rango de la medida o de su variación, por ejemplo). Esto sería una forma muy sencilla de redundancia software. Supóngase ahora que el sensor, en algunas ocasiones, puede estar perturbado por alguna interferencia exterior, si repetimos los mismos cálculos en diferentes momentos para tratar de evitar los efectos de la interferencia, el sistema será redundante en el tiempo. Si para evitar esas mismas perturbaciones

Departamento de Informática Universidad de Valladolid

disponemos de dos sensores, o bien si se instala otro procesador en previsión de que el primero pudiera fallar, el sistema dispondrá de redundancia hardware. En ese sistema de control también puede suceder que la comunicación entre los convertidores y el procesador no se considere segura, debido por ejemplo a ruidos eléctricos; en este caso, será conveniente añadir algunos bits a la comunicación y emplear un código detector o corrector de errores, eso sería redundancia informacional.

En los siguientes apartados estudiaremos con más detalle cada uno de los tipos de redundancia enunciados más arriba.

6.5.1. Redundancia hardware

La duplicación física del hardware es seguramente la redundancia más frecuente en los sistemas digitales. Existen tres formas básicas de redundancia hardware: (Johnson, 1984):

- Las técnicas de redundancia **pasiva** explotan el concepto de **enmascaramiento de los fallos** (ver sección 6.4). El interés de estas técnicas radica en que no necesitan que se tome ninguna acción externa por parte del sistema o del operador.

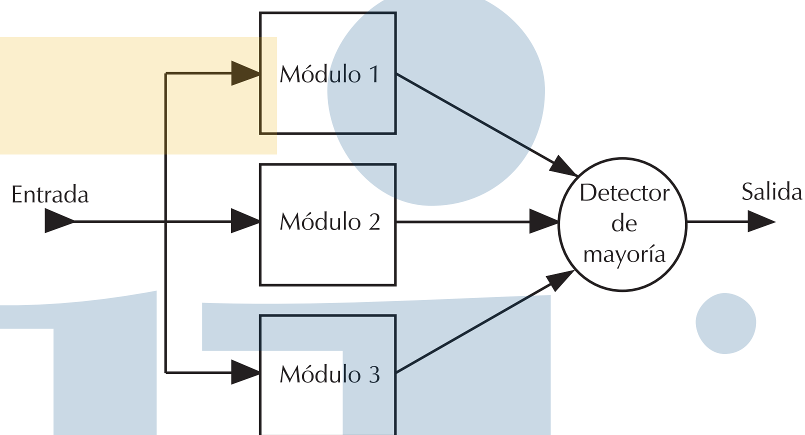
  Veremos a continuación un caso típico de redundancia pasiva denominado **redundancia modular triple** (TMR, *triple modular redundancy*). La construcción se muestra en la figura 6.6(a); un bloque básico en ella es el circuito detector de mayoría, cuya salida es la que sea mayoría entre las entradas. El circuito mostrado en la figura tiene como característica que es tolerante ante **un solo fallo**. Si se quiere que tenga mejores propiedades, puede ampliarse añadiendo más módulos, en ese caso la construcción recibe el nombre de redundancia n-modular (NMR, *n-modular redundancy)*. Un inconveniente del circuito de la figura 6.6 es que no es tolerante a fallos en el detector de mayoría. Esto puede corregirse incorporando también redundancia en dicho detector, como se indica en la figura 6.6(b). Una característica importante de este circuito es que proporciona tres salidas correctas aunque una entrada no lo sea (salvo que falle alguno de los detectores de mayoría), por esta razón, a este circuito se le llama **órgano de restauración**. En este último diseño, las salidas deben entrar en sucesivos circuitos redundantes, tal como se muestra en la figura 6.6(c).

  Un inconveniente de la redundancia modular triple (o n-modular) es que, al final de todos los cálculos, hay que dar un único resultado y no tres (o n). Este hecho implica que, al menos al final de todo el proceso, debe haber algún órgano sin redundancia (normalmente un detector de mayoría), lo que le hace sensible a fallos.

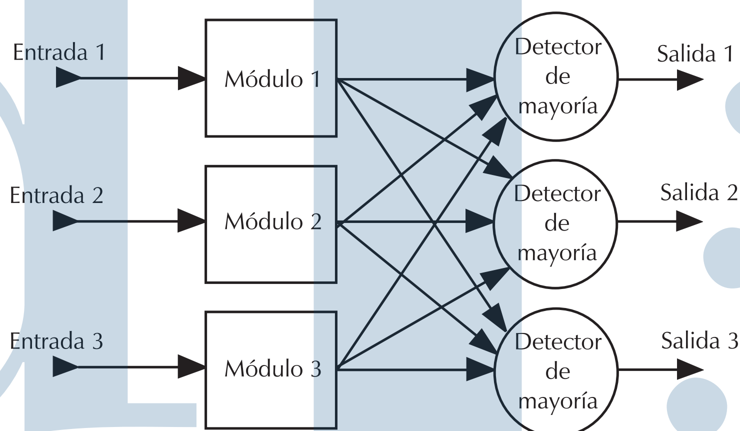
- Con el enfoque **activo** o **dinámico** se consigue la tolerancia a fallos detectando el error y ejerciendo alguna acción correctora para sustituir el órgano erróneo por otro alternativo. Como puede verse, las técnicas de redundancia activa precisan que el sistema se **reconfigure** para continuar funcionando correctamente. Evidentemente, la nueva configuración del sistema tardará un tiempo en establecerse, por lo que un sistema con redundancia activa debe permitir un funcionamiento incorrecto durante un corto periodo de tiempo, si bien esta situación será muy poco frecuente.

  En caso de fallo en un sistema con redundancia activa, son necesarias las siguientes fases: **detección**, **localización** y **recuperación** del fallo. Muchas veces estos conceptos no

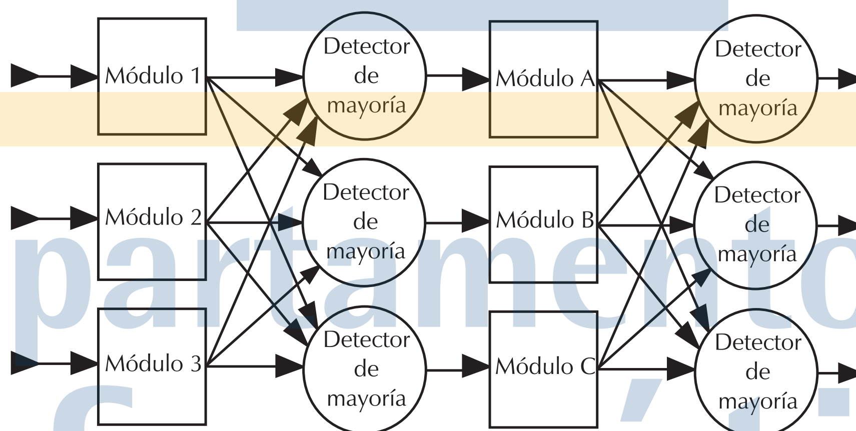
Departamento de Informática Universidad de Valladolid



(a)



(b)



(c)

Fig. 6.6. (a) Redundancia modular triple (TMR), (b) TMR tolerante a fallos en el detector de mayoría y (c) TMR en cadena.

Departamento de Informática
Universidad de Valladolid

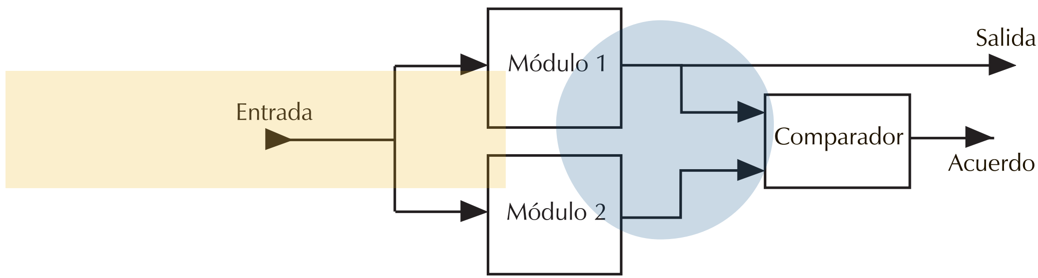


Fig. 6.7. Principio de funcionamiento del método de duplicación con comparación.

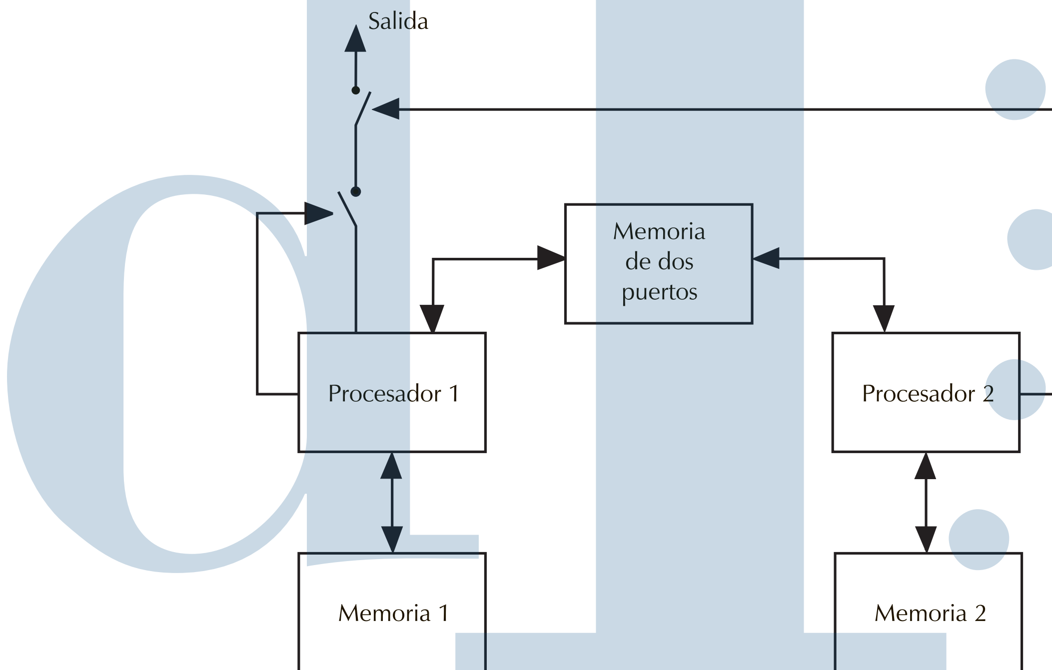


Fig. 6.8. Método de duplicación con comparación aplicado por software.

Departamento de Informática Universidad de Valladolid

están totalmente separados, por ejemplo, es posible detectar y localizar un fallo al mismo tiempo.

Un método sencillo de redundancia activa, que es capaz de detectar un fallo, pero no de localizarlo, es el denominado **duplicación con comparación**. Un esquema del principio de funcionamiento de esta técnica se muestra en la figura 6.7. Uno de los defectos de esta técnica es que no detecta los fallos en el comparador. Una adaptación interesante de la idea anterior consiste en efectuar la comparación por software, duplicando los procesadores. Esta idea se muestra esquemáticamente en la figura 6.8. El proceso de cálculo se efectúa en ambos procesadores dejando sendos resultados en la memoria de doble puerto. Después, los procesadores ejecutarán un software que comparará ambos resultados, por lo que un incorrecto funcionamiento en cualquiera de los procesadores, e incluso en la memoria de doble puerto que no está duplicada, será detectado.

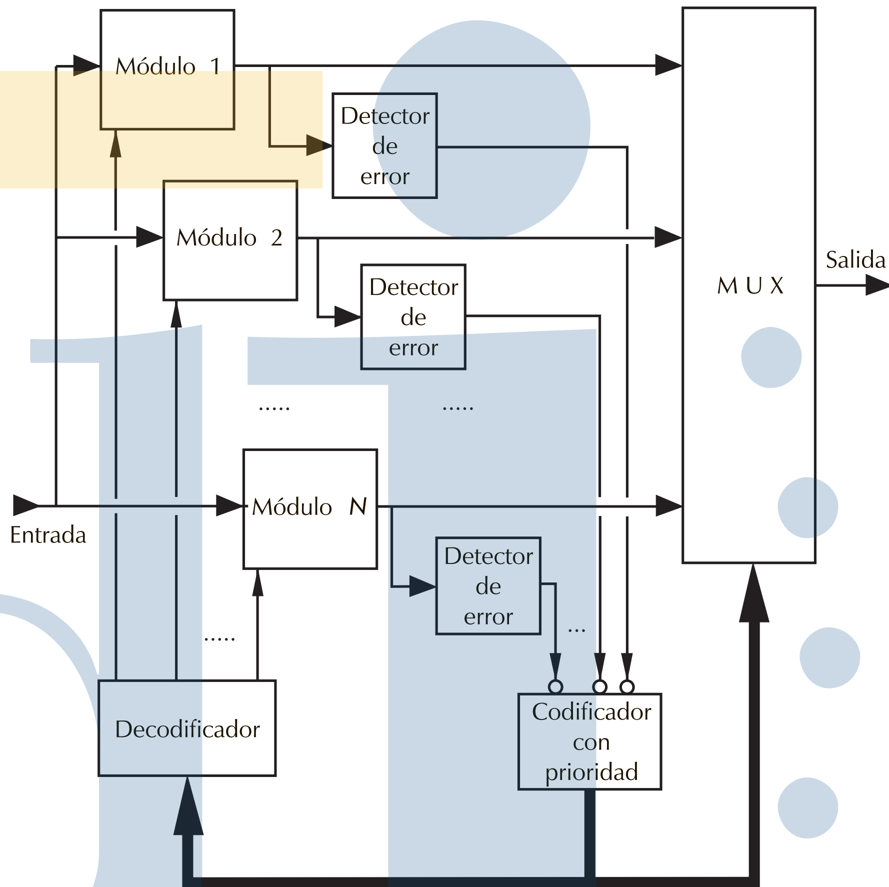


Fig. 6.9. Método de los repuestos en espera.

Otra técnica activa para detectar fallos son los **temporizadores centinelas**, (*watchdog timers*). Esta técnica consiste en un circuito, o programa, con un contador al que el sistema debe poner a cero periódicamente; en caso de no hacerlo el contador alcanzará un valor por encima de un límite establecido, por lo que se deducirá que el sistema tiene algún tipo de fallo.

Los temporizadores centinelas proporcionan una forma muy eficaz de detectar ciertos tipos de fallos. El caso más sencillo de fallo detectable es la parada total de un procesador. Otro hecho fácilmente detectable con los temporizadores centinelas es la sobrecarga de un procesador que haga que opere más lentamente; en este caso el temporizador detectará valores mayores que los normales en su contador. En general los temporizadores centinelas son eficaces para detectar cualquier tipo de falta de respuesta, tanto hardware como software. Éste es el típico caso de una rutina que se quede en un bucle infinito, en que el fallo impedirá la puesta a cero del contador.

Este método es bastante sencillo, por lo que pueden ponerse muchos temporizadores centinelas para acotar mejor la localización de un fallo.

Otro método de redundancia activa es el **repuesto en espera** que se ilustra en la figura 6.9. Con esta técnica uno de los módulos es el que funciona en cada momento y los demás

Departamento de Informática Universidad de Valladolid

están en reserva. Los detectores de error (que pueden ser temporizadores centinelas), a través del codificador, seleccionan la entrada activa del multiplexor. Este método tiene el inconveniente de que no es tolerante a fallos en el multiplexor. Esta técnica puede aplicarse de dos formas:

- En **frío**, es decir, los módulos de reserva no están operativos hasta que son seleccionados por producirse algún fallo en el módulo activo. Esto implica que el transitorio de conmutación de módulo será más lento, porque será necesario poner en funcionamiento el módulo de reserva. Este método se emplea en aplicaciones donde el consumo de energía es crítico y, sin embargo, puede permitirse un tiempo de transición más largo; un ejemplo de ello pueden ser algunos computadores instalados en satélites o sistemas portátiles alimentados por baterías.
- En **caliente**, que significa que los módulos de reserva están continuamente en funcionamiento. De esta forma la conmutación de módulo puede hacerse de manera mucho más rápida. Este modo de proceder es conveniente en sistemas críticos en que no es admisible un tiempo de conmutación largo, y en los que el consumo de energía no sea un factor importante.

Una interesante ventaja de la técnica de los repuestos en espera es que, en un sistema que contenga N módulos idénticos, se puede proporcionar mucha tolerancia a fallos con un número de repuestos bastante inferior a N. Evidentemente, aunque sólo se dispusiera de un repuesto, éste podría sustituir a cualquiera de los N módulos. Los casos en que un sistema posee N módulos son muy frecuentes; piénsese, por ejemplo, en sistemas multiprocesadores o sistemas que tengan como periféricos varios discos iguales.

- Las métodos **híbridos**, como su nombre indica, tratan de aprovechar los beneficios de las técnicas anteriores. En los sistemas con redundancia híbrida se emplea el enmascaramiento de fallos pero el sistema también puede reconfigurarse en caso de fallo. De este modo, el sistema está doblemente protegido. Por todo ello, la redundancia híbrida sólo se utiliza en sistemas que exijan una alta seguridad en su funcionamiento ya que su costo es excesivo.

Un primer método de redundancia híbrida es la **redundancia n-modular con repuestos**. Esta técnica, que se ilustra en la figura 6.10, consiste en reforzar la redundancia n-modular con módulos de repuesto adicionales. La idea central de este método es la comparación de la salida de cada módulo con la salida del detector de mayoría: si no hay acuerdo entre ambas salidas, se marcará el módulo como averiado y se sustituirá por uno de los repuestos. El elemento más complejo en los sistemas redundantes de este tipo es la red de conmutación, ya que debe proporcionar la posibilidad de reemplazar cada uno de los módulos por cada uno de los repuestos; esto se consigue con una red similar a una central de conmutación telefónica. Evidentemente, la tolerancia a fallos de estos sistemas es mayor que la de los sistemas con redundancia n-modular sin repuestos. Para aclarar esta idea supongamos que en el sistema de la figura 6.10, $N = 3$ y $M = 1$ (redundancia modular triple con un repuesto). Si se produjera un fallo en alguno de los módulos, entraría en juego el repuesto, pero, si se produjera un segundo fallo, ese fallo sería enmascarado por la redundancia pasiva: en consecuencia, ese sistema con 4 módulos podría alcanzar la tolerancia ante 2 fallos, sin embargo, si sólo aplicáramos redundancia pasiva, para obtener tolerancia a 2 fallos necesitaríamos 5 módulos.

Departamento de Informática Universidad de Valladolid

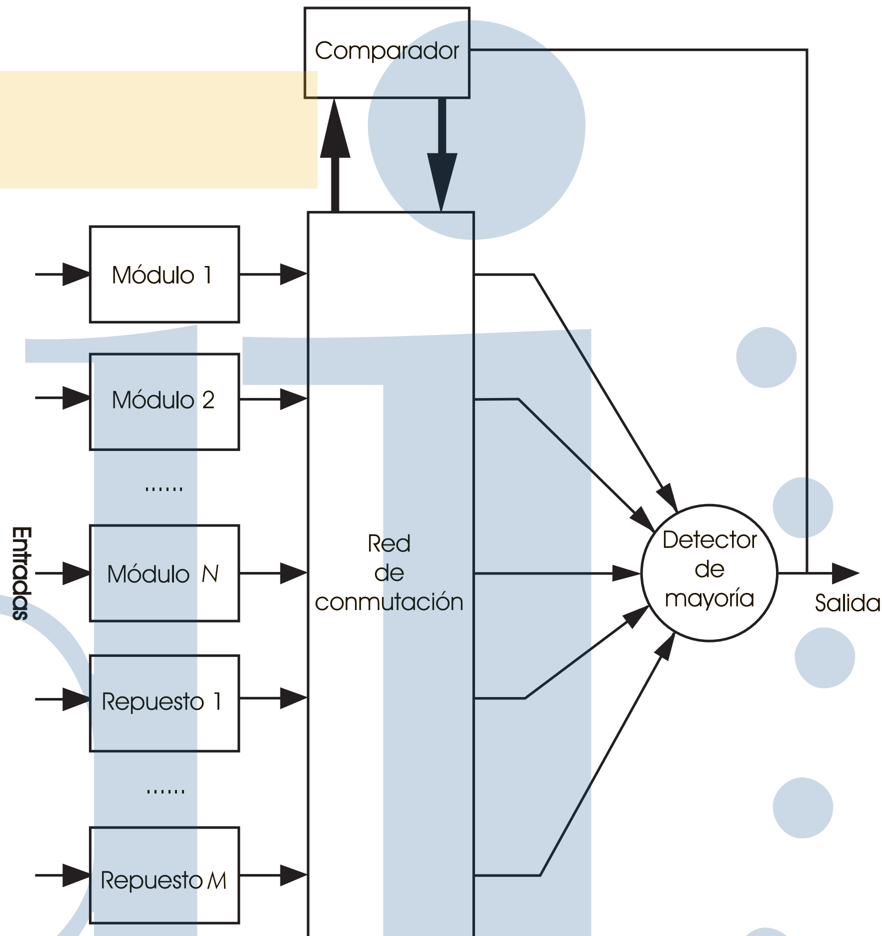


Fig. 6.10. Esquema de principio de la redundancia n-modular con repuestos.

Otro método híbrido es la **redundancia con autoeliminación** (***self-purging redundancy***) (Losq, 1976). La filosofía de este método es modificar la redundancia n-modular en el sentido de comparar la salida de cada módulo con la salida del detector de mayoría (véase la figura 6.11). En caso de desacuerdo, ese módulo se descartará de la votación. Por ello, el detector de mayoría deberá tener noticia del desacuerdo para invalidar dicha entrada. La diferencia de este método con el anterior radica en que, en este caso, no hay repuestos. Como puede apreciarse, el sistema enmascara los fallos mediante el detector de mayoría; por otra parte, se reconfigura anulando los módulos averiados. Por todo esto, la redundancia de este sistema se considera híbrida.

Otro método híbrido es la **arquitectura triple-dúplex** que también es una variante de la redundancia n-modular. En concreto, este tipo de arquitectura es una mezcla entre la redundancia modular triple y la duplicación con comparación. El esquema de principio de la arquitectura triple-dúplex se muestra en la figura 6.12. La idea central consiste en emplear el método de duplicación con comparación para detectar fallos en los módulos. cuando una pareja de módulos no se pone de acuerdo en el resultado, se excluye del detector de mayoría de la redundancia modular triple.

Departamento de Informática Universidad de Valladolid

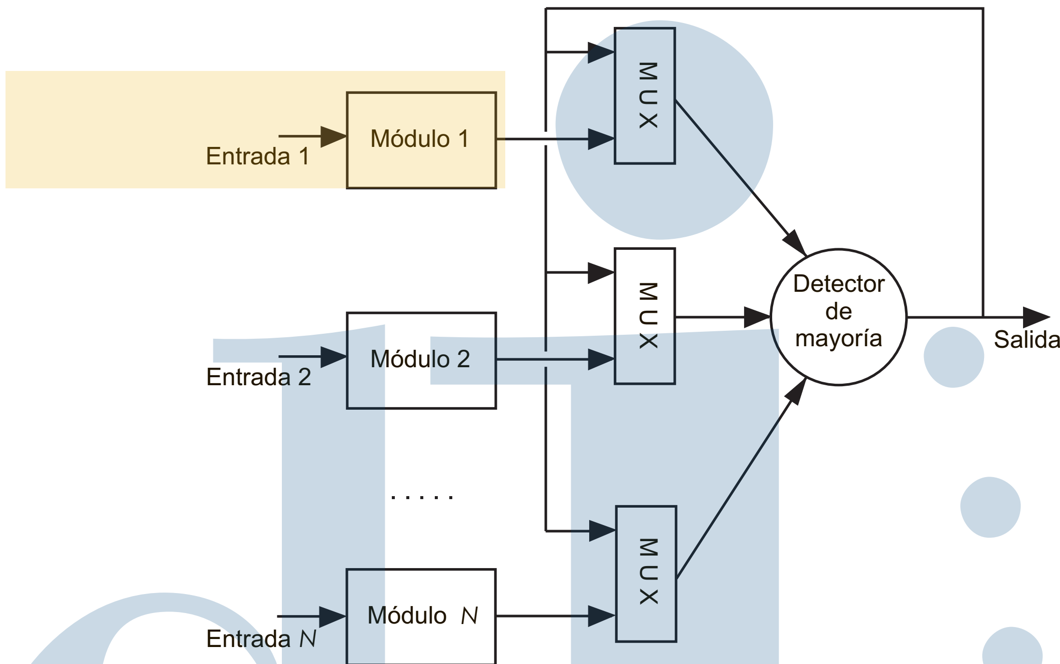


Fig. 6.11. Esquema de principio de la redundancia con autoeliminación.

6.5.2. Redundancia software

Para que un sistema posea este tipo de redundancia no es necesario duplicar los procedimientos que forman parte del programa; también se considera redundancia software el hecho de añadir código para efectuar funciones de comprobación que ayuden a detectar o corregir fallos. Las variantes de redundancia software son las siguientes:

- Las **pruebas de consistencia** que se basan en un conocimiento previo de algunas características de las informaciones para verificar si son correctas o no. Muchas pruebas de consistencia se basan sencillamente en comprobar si la magnitud sale o no fuera de unas **cotas** previamente establecidas. A veces también se establecen **cotas de variación**. Si la magnitud, o su variación, se salen fuera de esas cotas se puede deducir la existencia de algún error. Una de las magnitudes que las pruebas de consistencia pueden manejar es el rendimiento. Si midiendo éste por software se observa que baja de forma injustificada, será un indicio de fallo.

- Las **pruebas del hardware** consisten en habilitar procesos en el sistema que se dediquen a verificar el hardware. De esta forma, en un sistema con este tipo de redundancia, puede existir un proceso que escriba valores en las zonas de memoria que en cada momento no están utilizándose, y luego lea esas posiciones para comprobar que la memoria funciona correctamente; de igual manera, puede haber otros procesos que hagan pruebas sobre otros componentes del sistema.

Las técnicas de redundancia software analizadas hasta ahora son capaces de detectar por software errores o fallos del hardware. Los fallos de software se deben a errores en las

Departamento de Informática Universidad de Valladolid

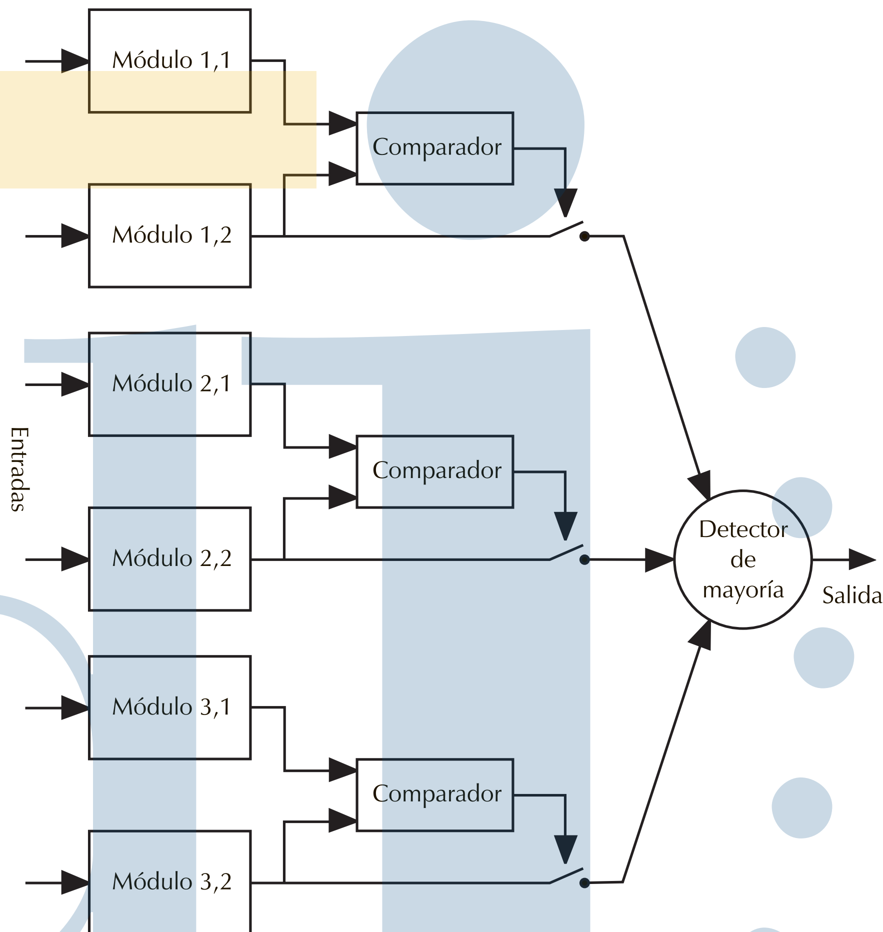


Fig. 6.12. Redundancia híbrida con arquitectura triple-dúplex.

especificaciones, la implementación o la codificación de dicho software. Este tipo de fallos sólo pueden detectarse si se duplica todo el proceso de construcción del software. En este línea va la siguiente forma de redundancia software:

- La **programación con** N **versiones** (Chen & Avizienis, 1978) consiste en diseñar y codificar los módulos de software N veces por N equipos de programadores, independientes unos de otros, y aplicar métodos similares a los de redundancia hardware pasiva (como la redundancia n-modular, por ejemplo). Con esta técnica se presupone que los diferentes equipos no cometerán las mismas equivocaciones. Sin embargo, esta suposición no siempre es cierta, en primer lugar porque las especificaciones dadas inicialmente a todos los equipos pueden ser erróneas, y, por otra parte, porque los diferentes equipos pueden incurrir en los mismos vicios de programación.

En cualquier caso, contra los fallos de software es más conveniente establecer medidas preventivas, tales como pruebas del software, controles de calidad, etc.

Departamento de Informática Universidad de Valladolid

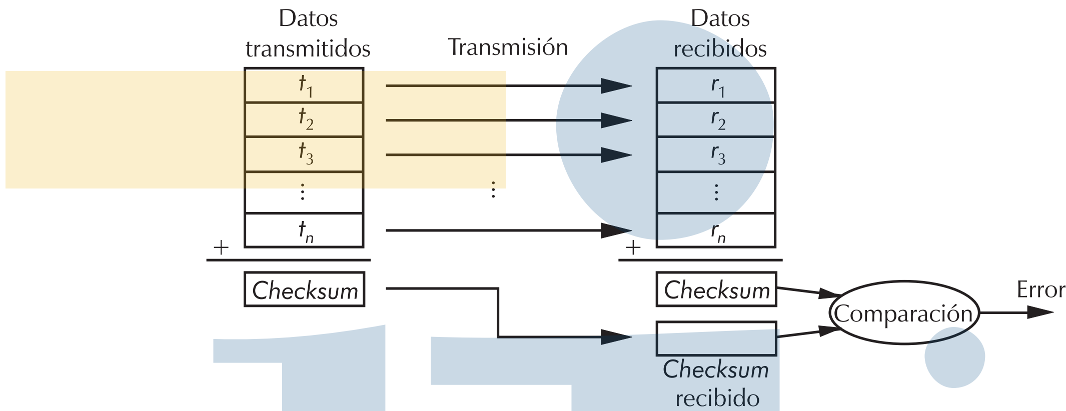


Fig. 6.13. Principio de funcionamiento de los *checksums*.

6.5.3. Redundancia informacional

Este tipo de redundancia consiste en añadir información adicional a los datos para permitir la detección o corrección de errores y se utiliza con mucha frecuencia cuando se almacena o se transmite información.

Un concepto bastante importante para la redundancia informacional es la **distancia de Hamming** entre dos palabras binarias que es el *número de bits en que ambas palabras difieren*. A partir de esta definición, se define la **distancia de Hamming de un código** como la *mínima distancia de Hamming entre dos palabras válidas del código*. En general, un código puede corregir hasta c bits y detectar hasta d errores en otros tantos bits si, y sólo si, se cumple:

$$2c + d + 1 \leq H_d$$

donde H_d es la distancia de Hamming del código.

Una forma simple de redundancia informacional son los **códigos con paridad**, que pueden ser **horizontales** o **verticales**. En el primer caso se añade el bit de paridad a cada palabra, y en el segundo se añade una palabra con los bits de paridad correspondientes a todos los bits de un mismo orden de las diferentes palabras en un chip, bloque, etc.

Los ***checksums*** son formas de redundancia informacional que se utilizan frecuentemente cuando se transmiten o almacenan bloques. El *checksum* consiste en un dato adicional que se agrega al bloque para detectar fallos o errores; este dato suele ser la suma de todos los datos del bloque o, a veces, una parte de la misma. La idea central de los *checksums* se muestra en la figura 6.13: el *checksum* se almacena junto al bloque, cuando éste se transmite, o lee, se recalcula el *checksum*: si ambos *checksums* no coinciden estaremos en presencia de algún tipo de fallo. Existen diferentes tipos de *checksums*, en función de la forma en que se calculen, los más frecuentemente empleados son los siguientes:

***Checksum* de simple precisión**: para calcularlo se suman todos los datos del bloque y se ignoran todas las llevadas que superen la longitud del dato. El problema de los *checksums* de simple precisión es que son incapaces de detectar algunos tipos de errores. Para ilustrar este hecho, tomemos el ejemplo de la figura 6.14; en este ejemplo se observa como el

Departamento de Informática Universidad de Valladolid

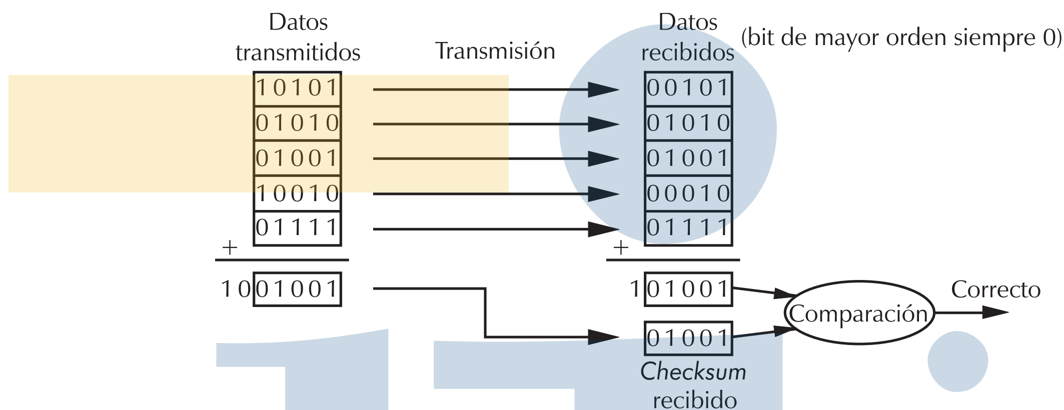


Fig. 6.14. Ejemplo en que se observa la imposibilidad de los *checksums* de simple precisión para detectar algunos fallos.

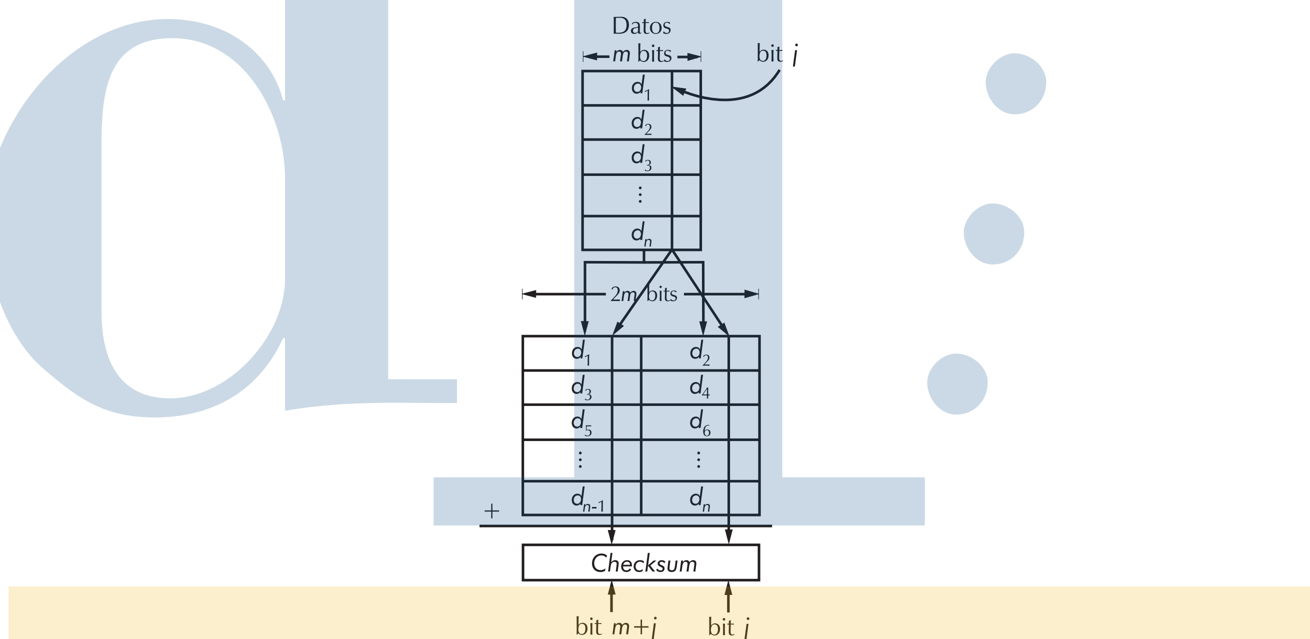


Fig. 6.15. Principio de funcionamiento del *checksum* de Honeywell.

checksum original y el calculado después de la transmisión son iguales, a pesar de existir un fallo en una de las líneas de transmisión (concretamente la línea correspondiente al bit de mayor peso está permanentemente puesta a 0).

***Checksum* de doble precisión**, en que para datos de n bits se calcula un *checksum* de $2n$ bits. De esta forma, si el bloque tiene un tamaño menor o igual que 2^n, no tendrá las limitaciones de los *checksums* de simple precisión y detectará la existencia de un error en todos los casos; si el bloque fuera mayor, la probabilidad de que algún error no pudiera detectarse, comenzará a crecer.

Departamento de Informática Universidad de Valladolid

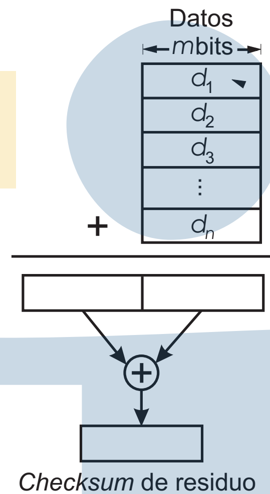


Fig. 6.16. Principio de funcionamiento del *checksum* de residuo.

***Checksum* de Honeywell**: la idea central de este tipo de *checksums* consiste en concatenar parejas de datos consecutivos para formar palabras de doble longitud. Una vez hecho esto, se calcula el *checksum* de simple precisión correspondiente a esas palabras de doble longitud, así el *checksum* tendrá también doble longitud. De esta forma lo que se busca es que un error en un bit fijo de los datos originales tenga efecto sobre dos bits del *checksum*, así será muy poco probable que un fallo no quede detectado. Un esquema del funcionamiento de esta forma de redundancia informacional se muestra en la figura 6.15, en que puede verse que un fallo en el bit j de la línea de transmisión se manifestará en los bits j y $m + j$ del *checksum*.

***Checksum* de residuo**: este tipo de *checksum* se calcula sumando todos los datos del bloque. Sin embargo, a diferencia del *checksum* de simple precisión, en lugar de ignorar la parte alta de la suma, se suma con la parte baja. De esta forma se obtiene un *checksum* del mismo tamaño que los datos, pero que es más efectivo que el de simple precisión.

Es necesario señalar que los *checksums* sólo pueden detectar errores, pero no localizarlos ni corregirlos.

Existen otras formas de redundancia informacional que no se describirán en detalle aquí, como los códigos cíclicos o aritméticos.

6.5.4. Redundancia temporal

Todos los métodos anteriores tienen el inconveniente de que precisan hardware o software adicionales, con el consiguiente incremento del costo. La redundancia temporal supone emplear un pequeño software que repita los cálculos en diferentes momentos a costa de emplear más tiempo. Se deberá recurrir a la redundancia temporal cuando el tiempo de ejecución no sea crítico y, sin embargo, haya limitaciones en el hardware.

La redundancia temporal puede ser especialmente efectiva para la **detección de fallos tran-**

Departamento de Informática
Universidad de Valladolid

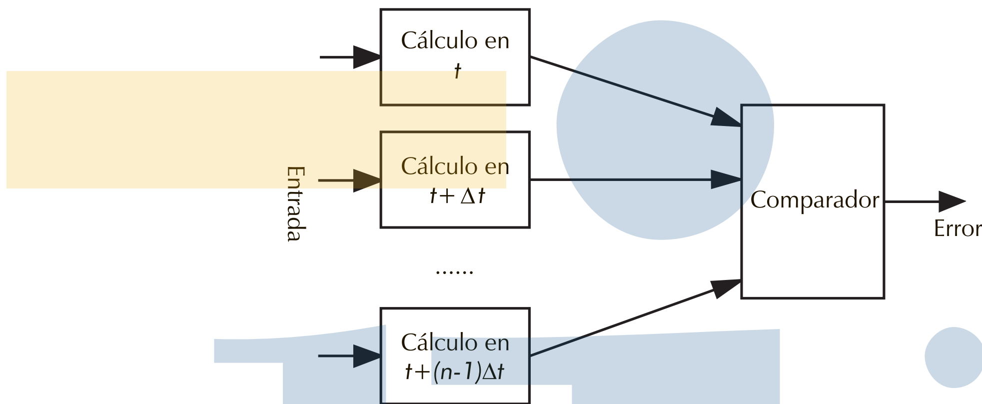


Fig. 6.17. Esquema de un sistema capaz de detectar fallos transitorios mediante redundancia temporal.

sitorios. Una idea válida para detectar fallos de este tipo se muestra en la figura 6.17. El principio de esta idea es la base de la redundancia temporal: efectuar el cálculo repetidamente en diferentes instantes y comparar los resultados. Si hubiera algún fallo transitorio, sólo afectará a uno de los cálculos y no a los demás, por lo que el comparador detectará el error. Como puede verse, el tiempo necesario para efectuar la computación completa es $n\Delta t$, siendo Δt el tiempo necesario para efectuar uno de los cálculos. En el caso de detectarse el error, se puede repetir el proceso para tratar de restablecer el acuerdo entre todos los resultados.

Una de las técnicas de redundancia temporal empleada para la detección de fallos permanentes, es la **lógica alternativa** (Reynolds & Metze, 1978), que consiste en repetir la operación después de cierto tiempo pero complementando la entrada. Este método es útil cuando se transmiten datos o, en general, cuando se quiera efectuar redundancia sobre una **función autodual**, es decir, una función que verifique la propiedad

$$f(\overline{x}) = \overline{f(x)} \qquad [6.1]$$

Cuando se aplica la lógica alternativa a una función con esta propiedad, la segunda vez que se efectúa la función debe obtenerse el complemento de la primera salida. Si en alguno de los bits hubiera coincidencia entre las dos salidas, existiría un fallo.

Es interesante indicar que, a partir de cualquier función lógica, se puede construir una función autodual añadiendo una entrada suplementaria. Esta función se consigue de la siguiente forma:

$$f_A(s,x) = sf(x) + \overline{s}\overline{f(\overline{x})}$$

donde f es la función lógica original, x representa todas las entradas de dicha función, s es la entrada suplementaria y f_A es la función autodual. Puede comprobarse fácilmente que la función así construida cumple la propiedad 6.1.

Hay otras formas de implementar la redundancia temporal, como por ejemplo: **operandos desplazados** (Patel & Fung, 1982), **operandos intercambiados** (Johnson, 1984) y **recálculo con duplicación con comparación** (Johnson *et al.* , 1988).

Departamento de Informática Universidad de Valladolid

6.6. Métodos de evaluación de sistemas tolerantes a fallos

6.6.1. Función de fiabilidad y tasa de fallos

En esta sección veremos la forma de medir la calidad de los sistemas en lo referente a la tolerancia a fallos. Uno de los parámetros que pueden medir dicha calidad es la **tasa de fallos del sistema** que puede definirse como el *número esperado de fallos en ese sistema por unidad de tiempo* (Shooman, 1968). Trataremos de modelar matemáticamente la tolerancia a fallos de los sistemas, especialmente de los redundantes.

Para hacerlo, definiremos la función **fiabilidad de un sistema** ($R(t)$) como la *probabilidad de que el sistema funcione correctamente durante todo el intervalo* $[t_0, t]$ *supuesto que lo hace en el instante* t_0.

Supongamos que tenemos N componentes idénticos que funcionan correctamente en el instante t_0, sea $N_o(t)$ el número de componentes que siguen funcionando correctamente en el instante t y $N_f(t)$ el resto, es decir los que han fallado a lo largo del intervalo $[t_0, t]$). La fiabilidad de los componentes en el instante t vendrá dada por:

$$R(t) = \frac{N_o(t)}{N} = \frac{N_o(t)}{N_o(t) + N_f(t)} \qquad [6.2]$$

Análogamente, a la probabilidad de que un componente falle, la podríamos llamar "fallabilidad" y viene dada por

$$Q(t) = \frac{N_f(t)}{N} = \frac{N_f(t)}{N_o(t) + N_f(t)}$$

Evidentemente

$$R(t) + Q(t) = 1 \qquad [6.3]$$

Por lo que podemos escribir la fiabilidad como

$$R(t) = 1 - \frac{N_f(t)}{N}$$

Derivando con respecto al tiempo se obtiene:

$$\frac{dR(t)}{dt} = -\frac{1}{N}\frac{dN_f(t)}{dt}$$

Que también se puede escribir como:

$$\frac{dN_f(t)}{dt} = -N\frac{dR(t)}{dt} \qquad [6.4]$$

Si al primer miembro de esta expresión lo dividimos por $N_o(t)$, obtendremos la **función de riesgo**, **tasa de riesgo**, o **tasa de fallos** anteriormente mencionada, $z(t)$:

$$z(t) = \frac{1}{N_o(t)}\frac{dN_f(t)}{dt}$$

Aplicando la ecuación 6.4, esta tasa de fallos puede expresarse así:

$$z(t) = \frac{1}{N_o(t)}\frac{dN_f(t)}{dt} = \frac{-N}{N_o(t)}\frac{dR(t)}{dt}$$

Departamento de Informática Universidad de Valladolid

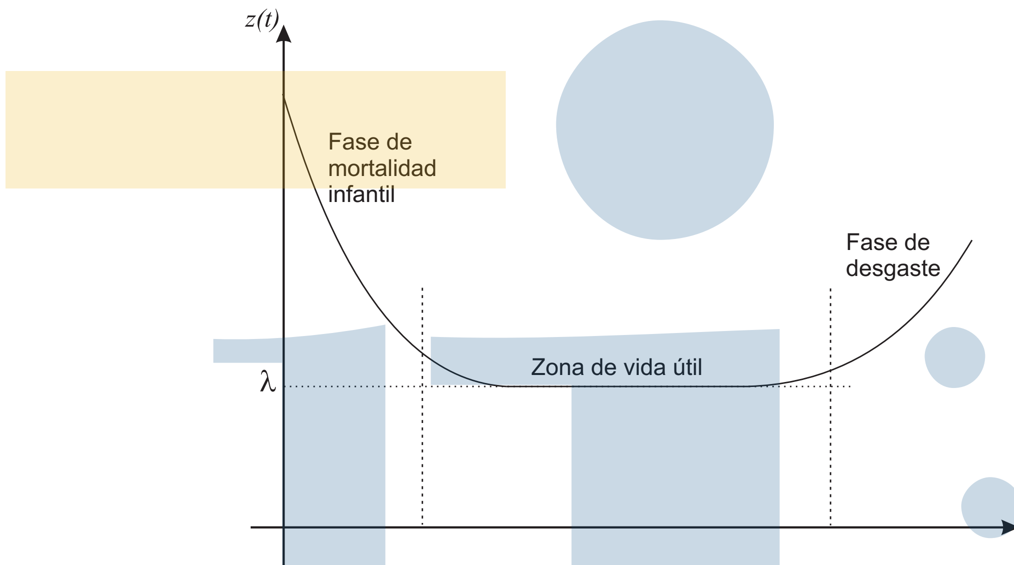


Fig. 6.18. Representación gráfica de la función de riesgo $z(t)$ en función del tiempo (curva de la bañera).

que, teniendo en cuenta la expresión 6.2, puede escribirse en función de la fiabilidad, $R(t)$:

$$z(t) = -\frac{1}{R(t)}\frac{dR(t)}{dt} \qquad [6.5]$$

Poniéndola en función de la fallabilidad, queda:

$$z(t) = \frac{1}{1 - Q(t)}\frac{dQ(t)}{dt}$$

Donde a la derivada de $Q(t)$ se le llama **densidad de fallos**.

La función de riesgo $z(t)$ dependiendo de la naturaleza del sistema tiene diferentes formas. La experiencia dice que para componentes electrónicos tiene la forma de **curva de bañera** que tiene tres zonas bien diferenciadas (figura 6.18):

- **Zona de mortalidad infantil**: en esta zona fallan los componentes inicialmente defectuosos.
- **Zona de estabilidad o de vida útil**: es la zona de mayor duración en que $z(t)$ permanece sensiblemente constante en un valor λ.
- **Zona de desgaste**: A partir de cierto momento los componentes han finalizado su vida útil y empiezan a fallar por desgaste.

Vamos a estudiar la curva en la zona de vida útil:

Según se desprende de la ecuación 6.5:

$$\frac{dR(t)}{dt} = -z(t)R(t) \qquad [6.6]$$

Departamento de Informática Universidad de Valladolid

Esta ecuación diferencial en $R(t)$ será lineal en la zona de vida útil de la curva de la bañera y quedará como:

$$\frac{dR(t)}{dt} = -\lambda R(t)$$

Esta ecuación diferencial tiene como solución:

$$R(t) = e^{-\lambda t} \qquad [6.7]$$

Esta función de la fiabilidad se conoce como **ley exponencial de los fallos** y es válida para muchos tipos de sistemas, especialmente físicos.

Sin embargo, para sistemas de software, no se puede asumir que la tasa de fallos sea constante, ya que un fallo software una vez definitivamente arreglado probablemente no se reproducirá, por lo que la tasa será decreciente; también pudiera ocurrir, por el contrario, que la reparación de un fallo software provoque nuevos errores. Una forma de modelar este tipo de sistemas es la **distribución de Weibull** (Siewiorek & Swarz, 1982) que viene dada por:

$$z(t) = \alpha\lambda(\lambda t)^{\alpha-1}$$

Esta función puede ser creciente o decreciente dependiendo del valor de α. Si $\alpha > 1$, la tasa de fallos es creciente, si $\alpha < 1$ es decreciente y si $\alpha = 1, z(t) = \lambda$ y entonces tenemos el caso anterior. En este caso, la ecuación diferencial 6.6 toma la forma:

$$\frac{dR(t)}{dt} = -\alpha\lambda(\lambda t)^{\alpha-1}R(t)$$

Cuya solución viene dada por:

$$R(t) = e^{-(\lambda t)^{\alpha}}$$

Afortunadamente, la mayoría de los sistemas obedecen a la curva de la bañera y se puede asumir que, en la mayor parte de su vida, la tasa de fallos es constante y, por tanto,siguen la ley exponencial de los fallos.

6.6.2. Tiempo medio de fallo ($MTTF$), tiempo medio de reparación ($MTTR$) y tiempo medio entre fallos ($MTBF$)

Otra medida de la incidencia de los fallos en un sistemas es el **tiempo medio de fallo** (*mean time to failure*, $MTTF$) que es el tiempo esperado que el sistema operará antes de que ocurra el primer fallo. Por ejemplo, si tenemos N sistemas idénticos puestos en operación en $t = 0$ y medimos el tiempo que funciona cada sistema antes de fallar (esto es equivalente a que el mismo sistema funcione durante un periodo de tiempo N veces más largo. El valor medio de todos esos tiempos sería el $MTTF$. Si cada sistema i funciona durante un tiempo t_i, antes del primer fallo, el $MTTF$ vendrá dado por:

$$MTTF = \frac{\sum_{i=1}^{N} t_i}{N}$$

Departamento de Informática Universidad de Valladolid

El $MTTF$ puede calcularse de forma estadística, es decir, mediante la función de densidad de probabilidad con la fórmula:

$$E[X] = \int_{-\infty}^{\infty} x f(x) dx$$

donde $f(x)$ es la función de densidad de probabilidad.

En nuestro caso, esta fórmula se transformará en:

$$MTTF = \int_{0}^{\infty} t f(t) dt \qquad [6.8]$$

La integral va ahora de 0 a ∞ porque en este caso no tiene sentido hablar de tiempos negativos y la función de densidad de probabilidad será la función de densidad de fallos definida en el apartado anterior,

$$f(t) = \frac{dQ(t)}{dt}$$

con lo que la ecuación 6.8 podremos escribirla así:

$$MTTF = \int_{0}^{\infty} t \frac{dQ(t)}{dt} dt \qquad [6.9]$$

teniendo en cuenta la expresión 6.3, se verificará que:

$$\frac{dQ(t)}{dt} = -\frac{dR(t)}{dt}$$

por lo que la ecuación 6.9 puede también escribirse como:

$$MTTF = \int_{0}^{\infty} -t \frac{dR(t)}{dt} dt$$

que, integrando por partes, queda

$$MTTF = \int_{0}^{\infty} -t \frac{dR(t)}{dt} dt = [-tR(t) + \int R(t) dt]_0^{\infty}$$

Teniendo en cuenta que en $t = 0$, $tR(t) = 0$ y que para valores grandes de t (mucho antes de ∞), $R(t) = 0$:

$$MTTF = \int_{0}^{\infty} R(t) dt$$

Si la función de fiabilidad obedece a la ley exponencial dada por 6.7, es decir, si ya hemos pasado la fase de mortalidad infantil del sistema y la tasa de fallos es constante,

$$MTTF = \int_{0}^{\infty} e^{-\lambda t} dt = [-\frac{1}{\lambda} e^{-\lambda t}]_0^{\infty} = \frac{1}{\lambda}$$

Departamento de Informática Universidad de Valladolid

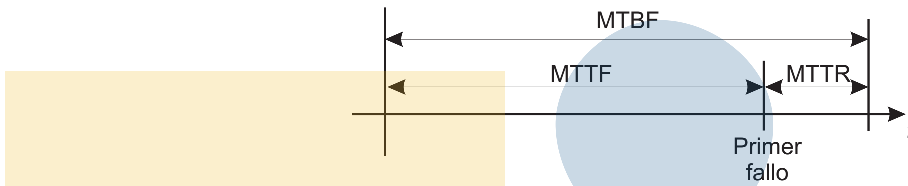


Fig. 6.19. Relación entre el $MTBF$, el $MTTF$ y el $MTTR$.

El **tiempo medio de reparación ($MTTR$)** (*Mean Time To Repair*) es sencillamente el *tiempo medio de reparación de un sistema cuando ocurre una avería*.

El **tiempo medio entre fallos** ($MTBF$) (*Mean Time Between Failures*) es el *tiempo medio entre dos averías consecutivas de un sistema*. Es interesante diferenciar entre el $MTBF$ y el $MTTF$, ya que este último es el tiempo que transcurre hasta la última avería.

Para calcular el $MTBF$, supongamos que N sistemas funcionan durante un tiempo T y que el número de fallos en el sistema i es n_i. El número medio de fallos por sistema es:

$$n_{medio} = \sum_{i=1}^{N} \frac{n_i}{N}$$

por tanto, el $MTBF$ es:

$$MTBF = \frac{T}{n_{medio}}$$

En la figura 6.19 se puede ver la relación entre el $MTBF$, el $MTTF$ y el $MTTR$, que es:

$$MTBF = MTTF + MTTR$$

Normalmente $MTTR$ es una pequeña fracción de $MTBF$ por lo que numéricamente $MTTF$ y $MTBF$ son muy similares.

Bibliografía y referencias

CHEN, L., & AVIZIENIS, A. 1978. N-version programming: A fault tolerant approach to realiability of software operation. *In: Proceedings of the International Symposium on Fault Tolerant Computing.*

JOHNSON, B.W. 1984. Fault-tolerant microprocessor-based systems. *IEEE micro*, **4**(6).

JOHNSON, B.W. 1989. *Design and Analysis of Fault Tolerant Digital Systems*. Addison Wesley.

JOHNSON, B.W., AYLOR, J.H., & HANA, H.H. 1988. Efficient use of time and hardware redundancy for concurrent error detection in a 32-bit VLSI adder. *IEEE Journal of Solid-State Circuits*, **23**(1).

KOREN, I., & MANI KRISHNA, C. 2007. *Fault-Tolerant Systems*. Morgan Kaufmann Publishers.

LOSQ, J. 1976. A highly efficient redundancy scheme: Self-purging redundancy. *IEEE Transactions on Computers*, **25**(6).

Departamento de Informática
Universidad de Valladolid

NELSON, V.P., & CARROL, B.D. 1982 (Nov.). Fault-tolerant computing (A tutorial). *In: AIAA Fault Tolerant Computing Workshop*.

PATEL, J.H., & FUNG, L.Y. 1982. Concurrent error detection in ALUs by recomputing with shifted operands. *IEEE Transactions on Computers*, **31**(7).

REYNOLDS, D.A., & METZE, G. 1978. Fault detection capabilities of alternating logic. *IEEE Transactions on Computers*, **27**(12).

SHOOMAN, M.L. 1968. *Probabilistic Reliability: An Engineering Approach*. McGraw-Hill.

SIEWIOREK, D.P., & SWARZ, R.S. 1982. *The Theory and Practice of Reliable System Design*. Digital Press.

CUESTIONES Y PROBLEMAS

6.1 Diseñar un circuito detector de mayoría de tres entradas digitales en un momento dado.

6.2 Ampliar el diseño del problema anterior para cinco entradas.

6.3 Supóngase que el detector de mayoría diseñado en el problema anterior se emplea para efectuar el escrutinio entre cinco entradas consistentes, cada una de ellas, en una secuencia de bits. Adaptar el diseño para que pueda funcionar correctamente aunque las entradas no estén exactamente sincronizadas (esto significa que los relojes que gobiernan las entradas están esencialmente sincronizados pero puede haber pequeños desfases entre ellos). Se puede suponer que se tiene acceso a los relojes de las entradas.

6.4 Demostrar que el sistema mostrado en la figura 6.8 funciona correctamente si ocurre **un solo fallo**, aunque sea en uno de los interruptores que controlan la salida

6.5 Diseñar de forma resumida la red de conmutación de un sistema con redundancia modular triple con dos repuestos.

Departamento de Informática Universidad de Valladolid