dit
UPM

Fiabilidad y tolerancia de fallos

Juan Antonio de la Puente
DIT/UPM

Transparencias basadas en el capítulo 5 del libro de A. Burns y A. Wellings *Real-Time Systems and Programming Languuages*, 3ª edición (2001)

Objetivos

- Veremos cuáles son los factores que afectan a la **fiabilidad** de un sistema
- También veremos algunas técnicas para **tolerar fallos** de software

©2001 Juan Antonio de la Puente

Índice

- Fiabilidad, averías y fallos
- Modos de fallo
- Prevención y tolerancia de fallos
- Redundancia estática y dinámica
  - Programación con N versiones
  - Bloques de recuperación
- Redundancia dinámica y excepciones
- Seguridad, fiabilidad y confiabilidad

©2001 Juan Antonio de la Puente

Fallos de funcionamiento

- Los fallos de funcionamiento de un sistema pueden tener su origen en
  - Una especificación inadecuada
  - **Errores de diseño del software**
  - Averías en el hardware
  - Interferencias transitorias o permanentes en las comunicaciones
- Nos centraremos en el estudio de los errores de software

©2001 Juan Antonio de la Puente

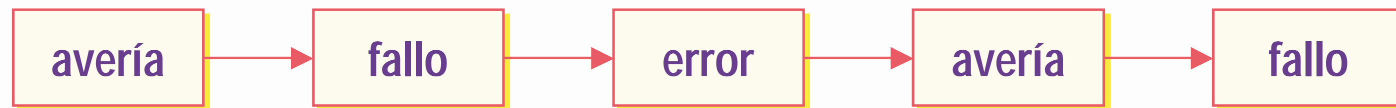
Conceptos básicos

- La **fiabilidad** (*reliability*) de un sistema es una medida de su conformidad con una especificación autorizada de su comportamiento
- Una **avería** (*failure*) es una desviación del comportamiento de un sistema respecto de su especificación
- Las averías se manifiestan en el comportamiento externo del sistema, pero son el resultado de **errores** (*errors*) internos
- Las causas mecánicas o algorítmicas de los errores se llaman **fallos** (*faults*)

©2001 Juan Antonio de la Puente

Fallos encadenados

- Los fallos pueden ser consecuencia de averías en los componentes del sistema (que son también sistemas)



©2001 Juan Antonio de la Puente

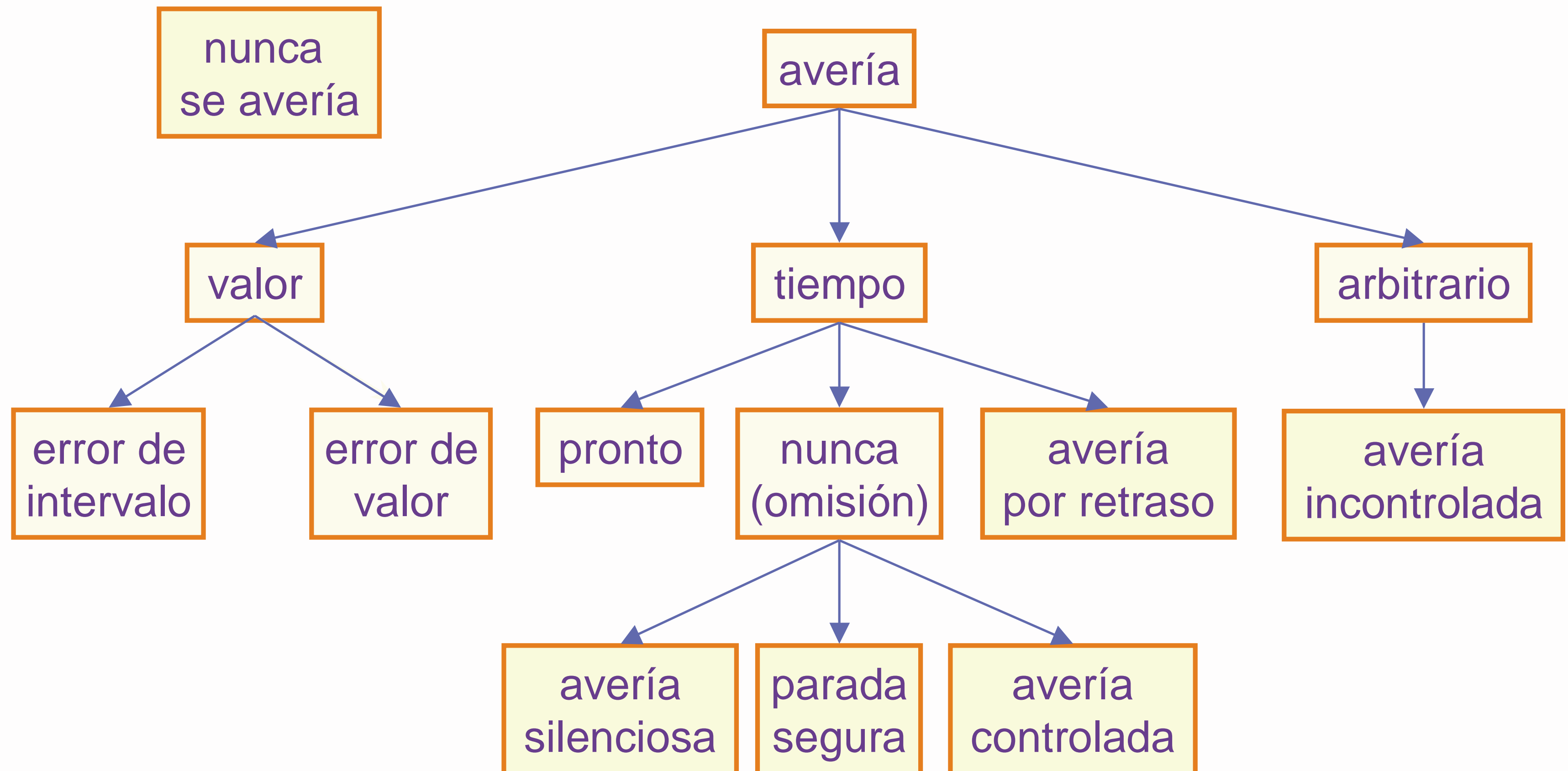
Tipos de fallos

- Fallos **transitorios**
  - desaparecen solos al cabo de un tiempo
  - ejemplo: interferencias en comunicaciones
- Fallos **permanentes**
  - permanecen hasta que se reparan
  - ejemplo: roturas de hardware, errores de diseño de software
- Fallos **intermitentes**
  - fallos transitorios que ocurren de vez en cuando
  - ejemplo: calentamiento de un componente de hardware

Debe impedirse que los fallos de todos estos tipos causen averías

©2001 Juan Antonio de la Puente

Tipos de avería *(failure modes)*



©2001 Juan Antonio de la Puente

Prevención y tolerancia de fallos

- Hay dos formas de aumentar la fiabilidad de un sistema:
  - **Prevención de fallos**
    - » Se trata de evitar que se introduzcan fallos en el sistema antes de que entre en funcionamiento
  - **Tolerancia de fallos**
    - » Se trata de conseguir que el sistema continúe funcionando aunque se produzcan fallos

- En ambos casos el objetivo es desarrollar sistemas con tipos de averías bien definidos

©2001 Juan Antonio de la Puente

Prevención de fallos

- Se realiza en dos etapas:
  - **Evitación de fallos**
    - Se trata de impedir que se introduzcan fallos durante la construcción del sistema
  - **Eliminación de fallos**
    - Consiste en encontrar y eliminar los fallos que se producen en el sistema una vez construido

©2001 Juan Antonio de la Puente

Técnicas de evitación de fallos

- **Hardware**
  - Utilización de componentes fiables
  - Técnicas rigurosas de montaje de subsistemas
  - Apantallamiento de hardware
- **Software**
  - Especificación de requisitos rigurosa o formal
  - Métodos de diseño comprobados
  - Lenguajes con abstracción de datos y modularidad
  - Utilización de entornos de desarrollo con computador (CASE) adecuados para gestionar los componentes

©2001 Juan Antonio de la Puente

Técnicas de eliminación de fallos

- **Comprobaciones**
  - Revisiones de diseño
  - Verificación de programas
  - Inspección de código
- **Pruebas** *(tests)*
  - Son necesarias, pero tienen problemas:
    - » no pueden ser nunca exhaustivas
    - » sólo sirven para mostrar que hay errores, no que no los hay
    - » a menudo es imposible reproducir las condiciones reales
    - » los errores de especificación no se detectan

©2001 Juan Antonio de la Puente

Limitaciones de la prevención de fallos

- Los componentes de hardware fallan,
  a pesar de las técnicas de prevención
  - La prevención es insuficiente si
    - » la frecuencia o la duración de las reparaciones es inaceptable
    - » no se puede detener el sistema para efectuar operaciones de mantenimiento

- Ejemplo: naves espaciales no tripuladas

- La alternativa es utilizar técnicas de **tolerancia de fallos**

©2001 Juan Antonio de la Puente

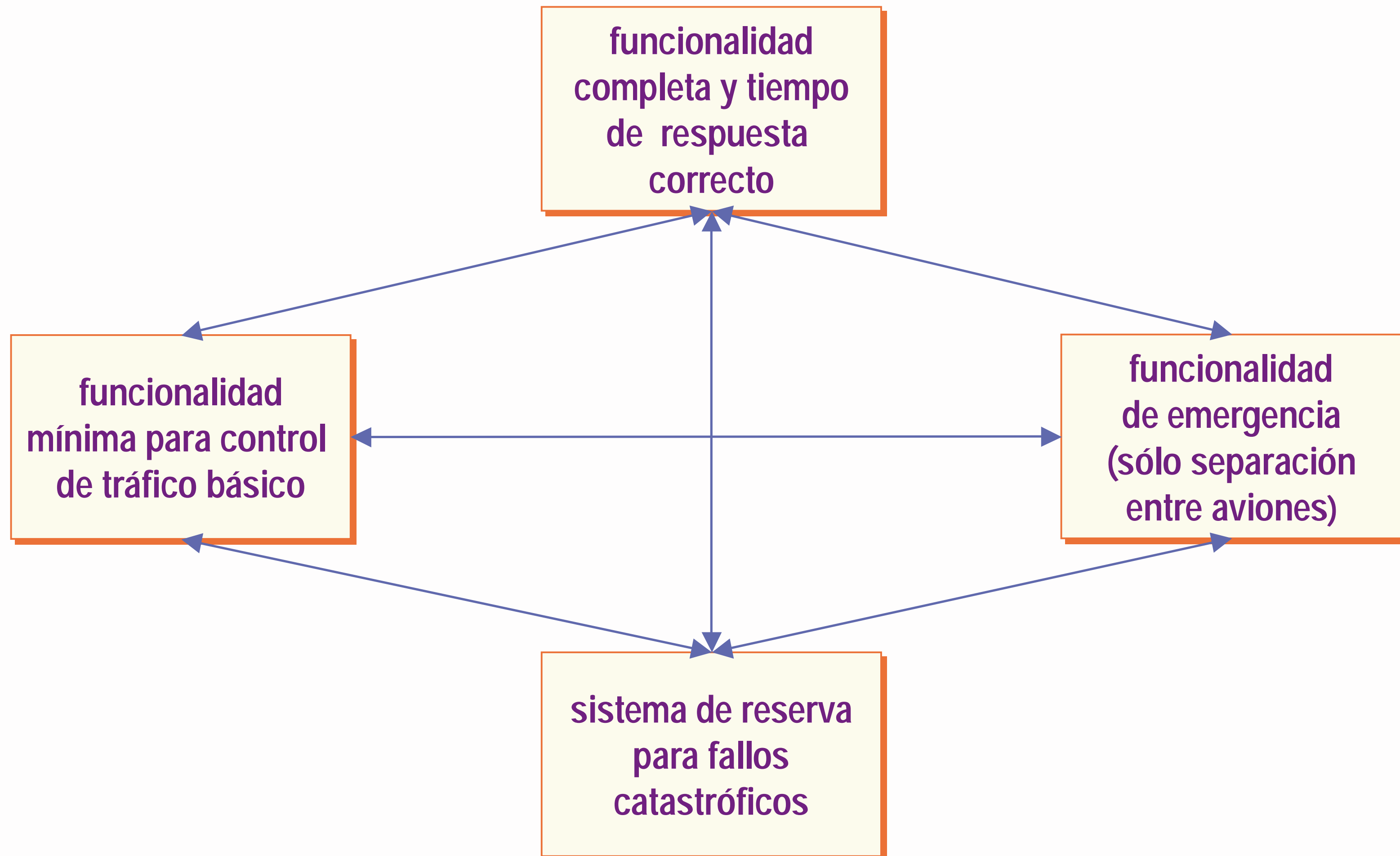
Grados de tolerancia de fallos

- **Tolerancia completa** *(fail operational)*
  - El sistema sigue funcionando, al menos durante un tiempo, sin perder funcionalidad ni prestaciones
- **Degradación aceptable** *(fail soft, graceful degradation)*
  - El sistema sigue funcionando con una pérdida parcial de funcionalidad o prestaciones hasta la reparación del fallo
- **Parada segura** *(fail safe)*
  - El sistema se detiene en un estado que asegura la integridad del entorno hasta que se repare el fallo

El grado de tolerancia de fallos necesario depende de la aplicación

©2001 Juan Antonio de la Puente

Ejemplo : control de tráfico aéreo



©2001 Juan Antonio de la Puente

Redundancia

- La tolerancia de fallos se basa en la **redundancia**
- Se utilizan **componentes adicionales** para detectar los fallos y recuperar el comportamiento correcto
- Esto aumenta la complejidad del sistema y puede introducir fallos adicionales
- Es mejor separar los componentes tolerantes del resto del sistema

©2001 Juan Antonio de la Puente

Redundancia en hardware

- **Redundancia estática**
  - Los componentes redundantes están siempre activos
  - Se utilizan para **enmascarar** los fallos
  - Ejemplo:
    - » Redundancia modular triple (ó N), TMR/NMR
- **Redundancia dinámica**
  - Los componentes redundantes se activan cuando se detecta un fallo
  - Se basa en la detección y posterior recuperación de los fallos
  - Ejemplos:
    - » sumas de comprobación
    - » bits de paridad

©2001 Juan Antonio de la Puente

Tolerancia de fallos de software

- Técnicas para detectar y corregir errores de diseño

- **Redundancia estática**
  - Programación con N versiones

- **Redundancia dinámica**
  - Dos etapas: **detección** y **recuperación** de fallos
  - Bloques de recuperación
    - » Proporcionan recuperación hacia atrás
  - Excepciones
    - » Proporcionan recuperación hacia adelante

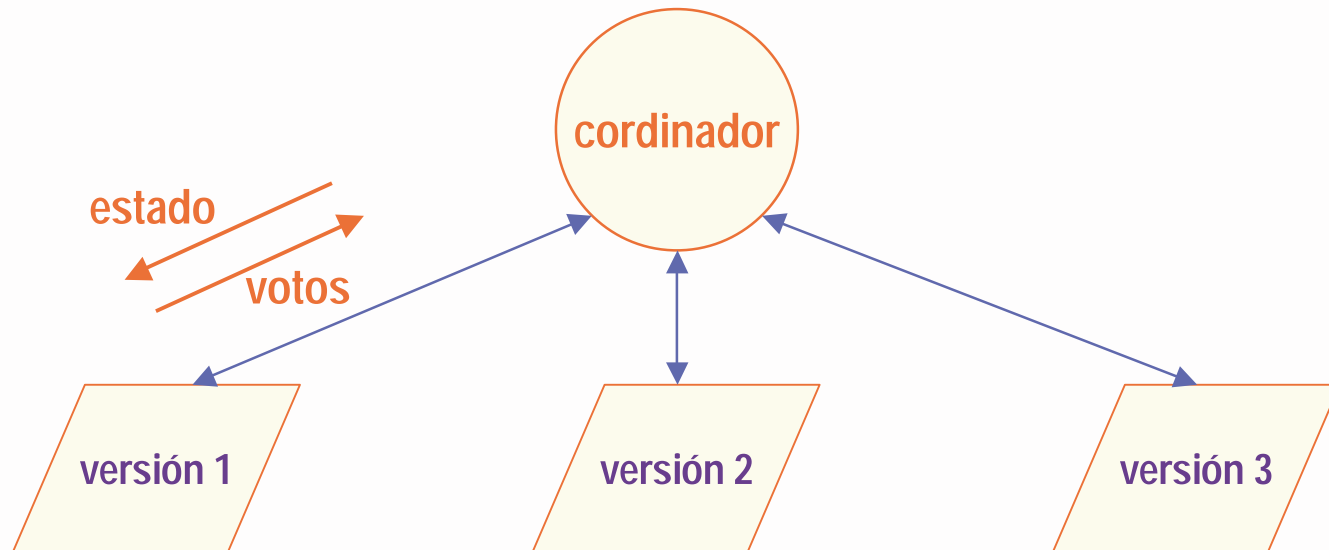
©2001 Juan Antonio de la Puente

Programación con N versiones

- **Diversidad de diseño**
  - N programas desarrollados independientemente con la misma especificación
  - sin interacciones entre los equipos de desarrollo
- **Ejecución concurrente**
  - proceso coordinador *(driver)*
    - intercambia datos con los procesos que ejecutan las versiones
  - todos los programas tienen las mismas entradas
  - las salidas se comparan
  - si hay discrepancia se realiza una **votación**

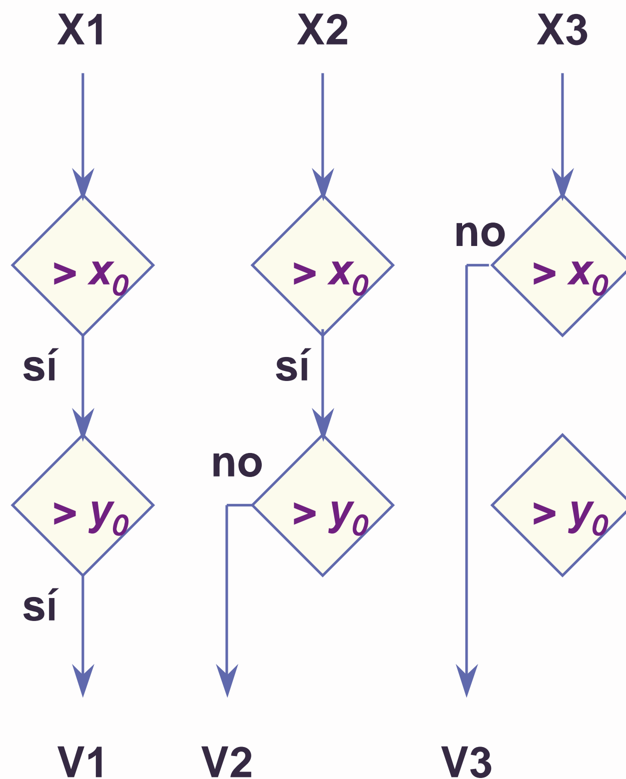
©2001 Juan Antonio de la Puente

Programación con N versiones



©2001 Juan Antonio de la Puente

Comparación consistente



- La comparación de valores reales o texto no es exacta
- Cada versión produce un resultado correcto, pero diferente de las otras
- No se arregla comparando con x0+Δ, y0+Δ

©2001 Juan Antonio de la Puente

Problemas

- La correcta aplicación de este método depende de:
  - **Especificación inicial**
    - » Un error de especificación aparece en todas las versiones
  - **Desarrollo independiente**
    - » No debe haber interacción entre los equipos
    - » No está claro que distintos programadores cometan errores independientes
  - **Presupuesto suficiente**
    - » Los costes de desarrollo se multiplican
      - ¿sería mejor emplearlos en mejorar una versión única?
    - » El mantenimiento es también más costoso
- Se ha utilizado en sistemas de aviónica críticos.

©2001 Juan Antonio de la Puente

Redundancia dinámica en software

Cuatro etapas:

1. **Detección** de errores
   » no se puede hacer nada hasta que se detecta un error
2. **Evaluación** y **confinamiento** de los daños
   » *diagnosis*: averiguar hasta dónde ha llegado la información errónea
3. **Recuperación** de errores
   » llevar el sistema a un estado correcto, desde el que pueda seguir funcionando (tal vez con funcionalidad parcial)
4. **Reparación** de fallos
   » Aunque el sistema funcione, el fallo puede persistir y hay que repararlo

©2001 Juan Antonio de la Puente

Detección de errores

- Por el **entorno de ejecución**
  - hardware (p.ej.. instrucción ilegal)
  - núcleo o sistema operativo (p.ej. puntero nulo)
- Por el **software de aplicación**
  - Duplicación (redundancia con dos versiones)
  - Comprobaciones de tiempo
    - *watchdog timer*
    - *deadline checks*
  - Inversión de funciones
  - Códigos detectores de error
  - Validación de estado
  - Validación estructural

©2001 Juan Antonio de la Puente

Evaluación y confinamiento de daños

- Es importante confinar los daños causados por un fallo a una parte limitada del sistema *(firewalling)*
- Se trata de estructurar el sistema de forma que se minimice el daño causado por los componentes defectuosos (compartimentos estancos, *firewalls*)
- Técnicas
  - **Descomposición modular**: confinamiento estático
  - **Acciones atómicas**: confinamiento dinámico

©2001 Juan Antonio de la Puente

Recuperación de errores

- Es la etapa más importante
- Se trata de situar el sistema en un estado correcto desde el que pueda seguir funcionando
- Hay dos formas de llevarla a cabo:
  - **Recuperación directa** (hacia adelante) *(FER)*
    - Se avanza desde un estado erróneo haciendo correcciones sobre partes del estado
  - **Recuperación inversa** (hacia atrás) *(BER)*
    - Se retrocede a un estado anterior correcto que se ha guardado previamente

©2001 Juan Antonio de la Puente

Recuperación directa

- La forma de hacerla es específica para cada sistema
- Depende de una predicción correcta de los posibles fallos y de su situación
- Hay que dejar también en un estado seguro el sistema controlado
- Ejemplos
  - punteros redundantes en estructuras de datos
  - códigos autocorrectores

©2001 Juan Antonio de la Puente

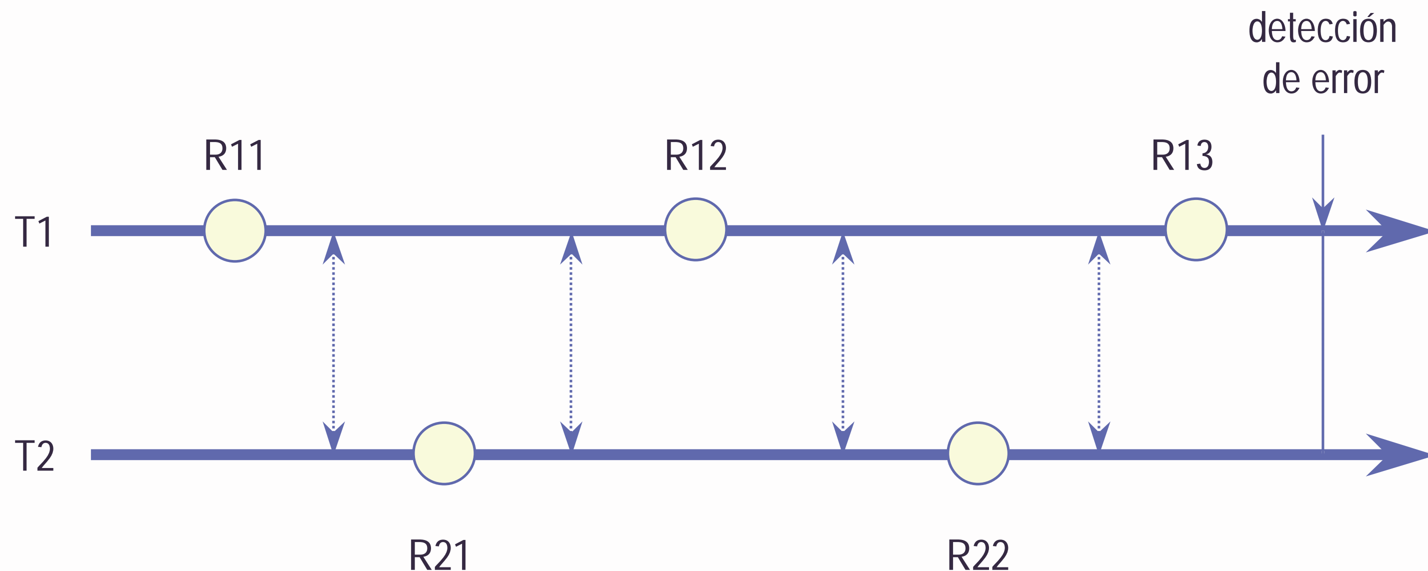
Recuperación inversa

- Consiste en retroceder a un estado anterior correcto y ejecutar un segmento de programa alternativo (con otro algoritmo)
  - El punto al que se retrocede se llama **punto de recuperación** *(recovery point)*
  - La acción de guardar el estado se llama *chekpointing*
- No es necesario averiguar la causa ni la situación del fallo
  - Sirve para fallos imprevistos
- ¡Pero no puede deshacer los errores que aparecen en el sistema controlado!

©2001 Juan Antonio de la Puente

Efecto dominó

- Cuando hay tareas concurrentes la recuperación se complica



- Solución: *líneas de recuperación* consistentes para todas las tareas

©2001 Juan Antonio de la Puente

Reparación de fallos

- La reparación automática es difícil y depende del sistema concreto
- Hay dos etapas
  - **Localización del fallo**
    - Se pueden utilizar técnicas de detección de errores
  - **Reparación del sistema**
    - Los componentes de hardware se pueden cambiar
    - Los componentes de software se reparan haciendo una nueva versión
    - En algunos casos puede ser necesario reemplazar el componente defectuoso sin detener el sistema

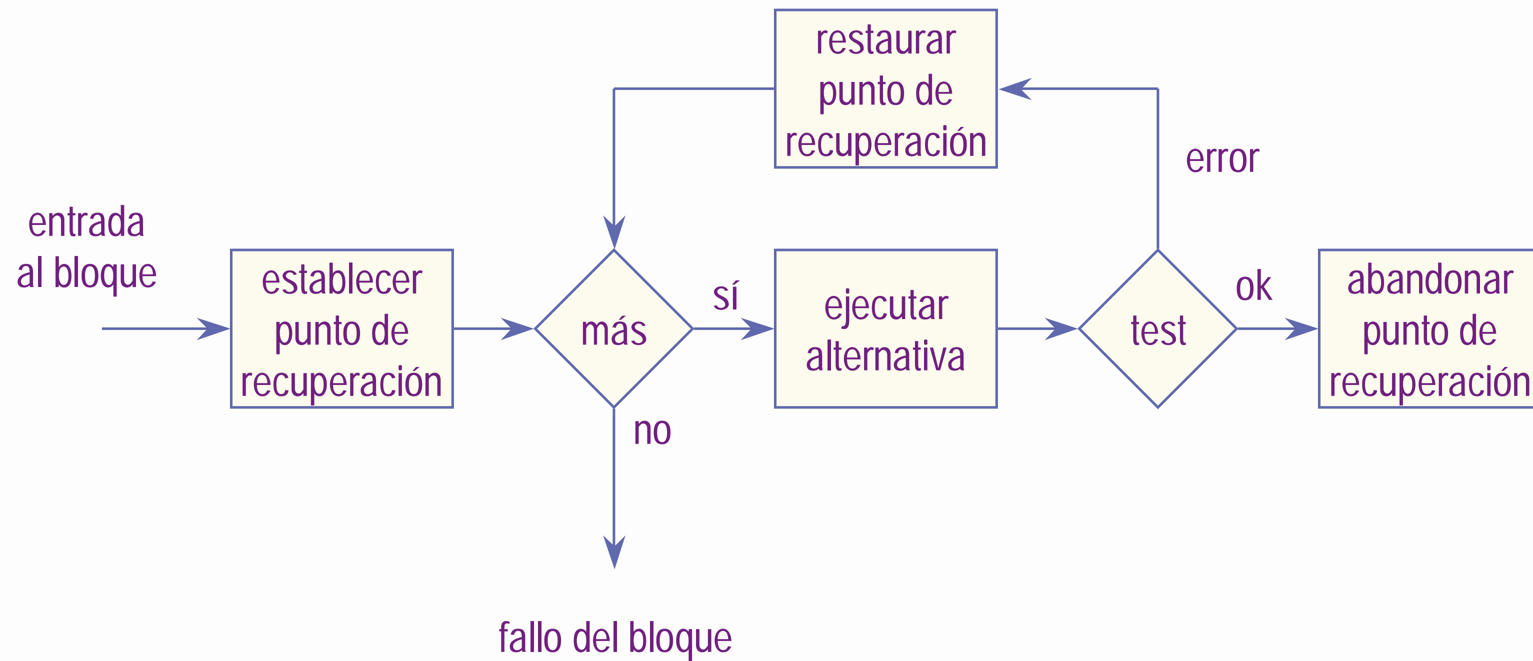
©2001 Juan Antonio de la Puente

Bloques de recuperación

- Es una técnica de recuperación inversa integrada en el lenguaje de programación
- Un **bloque de recuperación** es un bloque tal que
  - su entrada es un punto de recuperación
  - a su salida se efectúa una prueba de aceptación
    - sirve para comprobar si el módulo primario del bloque termina en un estado correcto
  - si la prueba de aceptación falla,
    - se restaura el estado inicial en el punto de recuperación
    - se ejecuta un **módulo alternativo** del mismo bloque
  - si vuelve a fallar, se siguen intentando alternativas
  - cuando no quedan más, el bloque falla y hay que intentar al recuperación en un nivel más alto

©2001 Juan Antonio de la Puente

Esquema de recuperación



©2001 Juan Antonio de la Puente

Sintaxis

```
ensure <condición de aceptación>
by
   <módulo primario>
else by
   <módulo alternativo>
else by
   <módulo alternativo>
...
else by
   <módulo alternativo>
else error;
```

Puede haber bloques anidados

- si falla el bloque interior, se restaura el punto de recuperación del bloque exterior

©2001 Juan Antonio de la Puente

Ejemplo: ecuación diferencial

```
ensure error <= tolerance
by
   Explicit_Runge_Kutta;
else by
   Implicit_Runge_Kutta;
else error;
```

- El método explícito es más rápido, pero no es adecuado para algunos tipos de ecuaciones
- El método implícito sirve para todas las ecuaciones, pero es más lento
- Este esquema sirve para todos los casos
- Puede tolerar fallos de programación

©2001 Juan Antonio de la Puente

Prueba de aceptación

- Es fundamental para el buen funcionamiento de los bloques de recuperación
- Hay que buscar un compromiso entre detección exhaustiva de fallos y eficiencia de ejecución
- Se trata de asegurar que el resultado es aceptable, no forzosamente correcto
- Pero hay que tener cuidado de que no queden errores residuales sin detectar

©2001 Juan Antonio de la Puente

Comparación

N versiones

- Redundancia estática
- Diseño
  - algoritmos alternativos
  - proceso coordinador
- Ejecución
  - múltiples recursos
- Detección de errores
  - votación

Bloques de recuperación

- Redundancia dinámica
- Diseño
  - algoritmos alternativos
  - prueba de aceptación
- Ejecución
  - puntos de recuperación
- Detección de errores
  - prueba de aceptación

¡Ambos métodos son sensibles a los errores en los requisitos!

©2001 Juan Antonio de la Puente

Excepciones y redundancia dinámica

- Una **excepción** es una manifestación de un cierto tipo de error
- Cuando se produce un error, se *eleva (raise, signal, throw)* la excepción correspondiente en el contexto donde se ha invocado la operación errónea
- Esto permite **manejar** la excepción en este contexto
- Se trata de un mecanismo de **recuperación directa** de errores (no hay vuelta atrás)
- Pero se puede utilizar para realizar recuperación inversa también

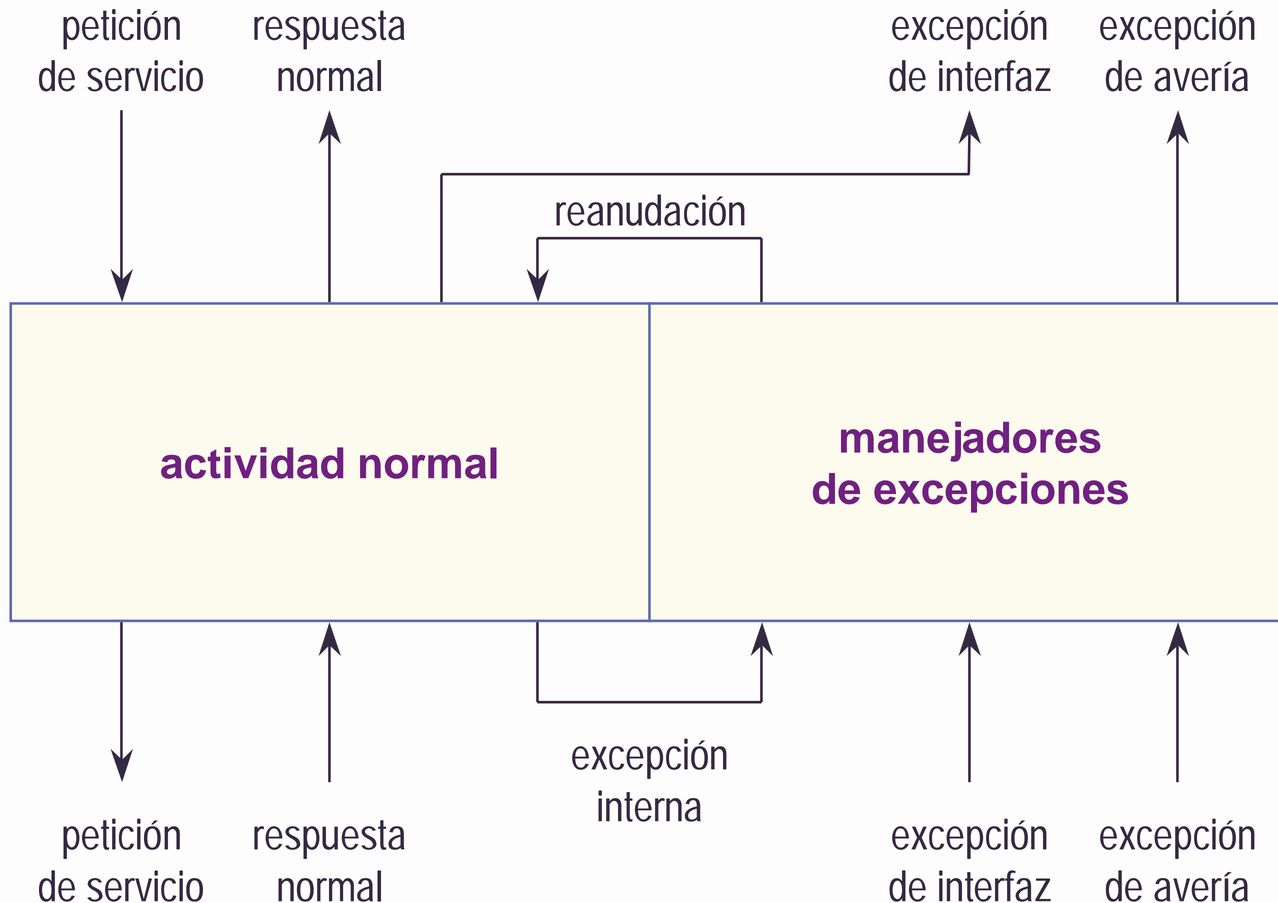
©2001 Juan Antonio de la Puente

Aplicaciones de las excepciones

- Tratar **situaciones anormales** en el entorno de ejecución
- **Tolerar fallos** de diseño de software
- Facilitar un **mecanismo generalizado** de **detección y corrección de errores**

©2001 Juan Antonio de la Puente

Componente ideal tolerante con los fallos



©2001 Juan Antonio de la Puente

Seguridad y fiabilidad

- Un sistema es **seguro** si no se pueden producir situaciones que puedan causar muertes, heridas, enfermedades, ni daños en los equipos ni en el ambiente
  - Un **accidente** *(mishap)* es un suceso (o una serie de sucesos) imprevisto que puede producir daños inadmisibles
- La **fiabilidad** es la probabilidad de que un sistema se comporte de acuerdo con su especificación
- La **seguridad** es la probabilidad de que no ocurra ningún suceso que provoque un accidente

¡Seguridad y fiabilidad pueden estar en conflicto!

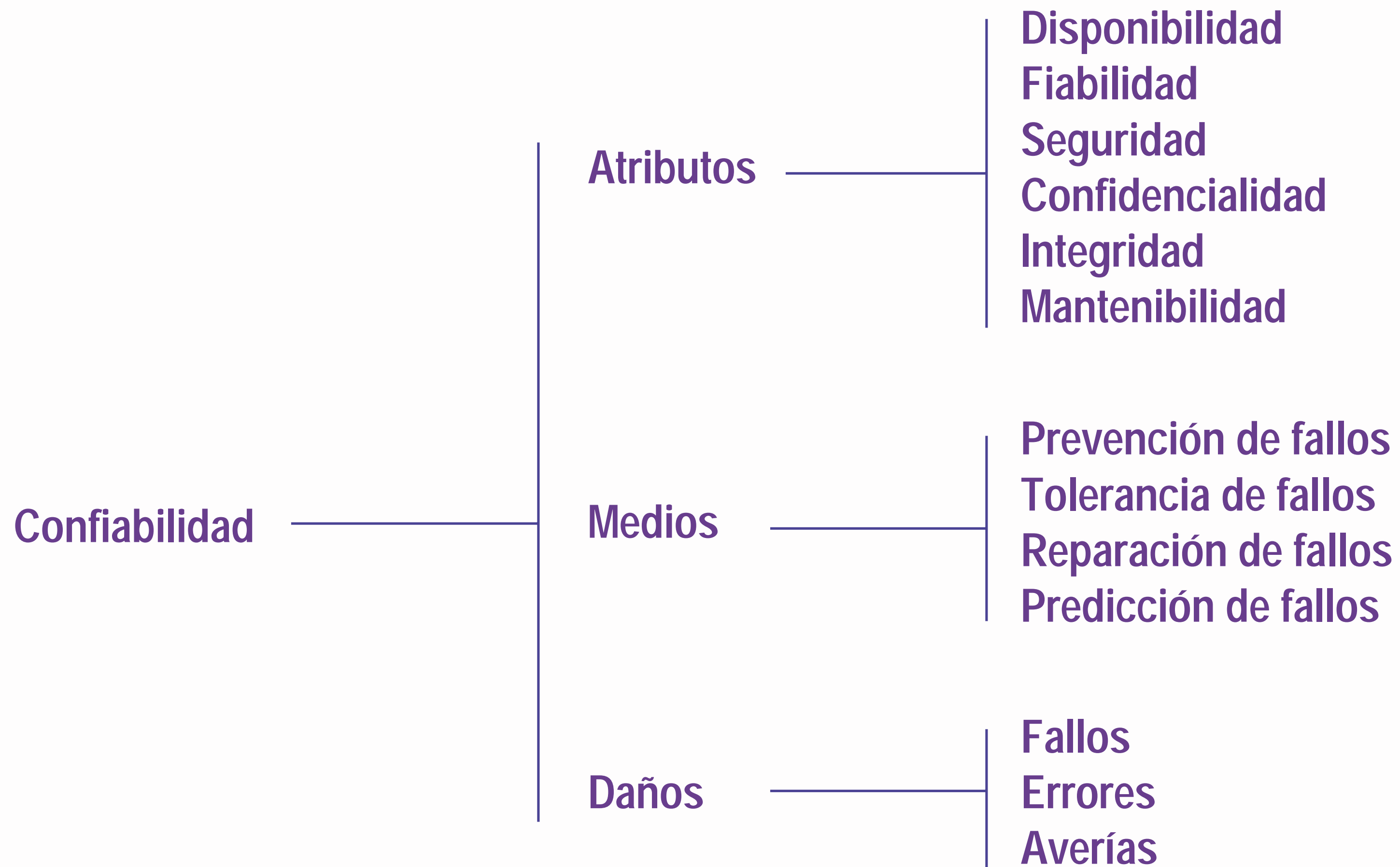
©2001 Juan Antonio de la Puente

Confiabilidad

- La **confiabilidad** *(dependability)* es una propiedad de los sistemas que permite confiar justificadamente en el servicio que proporcionan
- Tiene varios aspectos

©2001 Juan Antonio de la Puente

Terminología



©2001 Juan Antonio de la Puente

Resumen

- La **fiabilidad** de un sistema es una medida de su conformidad con una especificación autorizada de su comportamiento
- La fiabilidad de un sistema se puede aumentar mediante técnicas de **prevención** o de **tolerancia** de fallos
- La tolerancia de fallos se basa en la **redundancia**
  - **estática** (por ejemplo, N versiones)
  - **dinámica** (por ejemplo, bloques de recuperación)
- Las **excepciones** proporcionan redundancia dinámica con recuperación directa
- La **confiabilidad** de un sistema es una propiedad más amplia que la fiabilidad

©2001 Juan Antonio de la Puente