Desarrollo de Software para Sistemas Empotrados

1. Introducción
2. *Plataformas para sistemas empotrados*
3. Especificación y análisis de requisitos software en sistemas empotrados
4. Diseño arquitectónico en sistemas empotrados
5. Implementación software de sistemas empotrados

Desarrollo de Software para Sistemas Empotrados

2. Plataformas para sistemas empotrados

- Funcionamiento sobre máquina desnuda: ejecutivos cíclicos
- Sistemas operativos dirigidos por eventos
- Reserva de recursos
- Particionado en el espacio y el tiempo
- Particionamiento hardware-software
- Plataformas basadas en lenguajes de programación concurrentes
- Sistemas empotrados distribuidos

Tipos de plataformas, por su variabilidad (*jitter*) máxima

Tipo	Comentario	Max Jitter
HW específico: ASIC	Costoso salvo para muchas unidades (>400K), poco flexible	< 1μs
HW específico: FPGA	Coste y prestaciones medios, flexible	< 1μs
Máquina desnuda	Solo para sistemas muy simples	≅1μs
Sistema operativo de tiempo real	Ej: VxWorks, QNX, RTEMS, MaRTE OS	≅10μs
Sistema operativo adap-tado para tiempo real	Ej: Linux de tiempo real	≅1ms
Sistema operativo de pro-pósito general	Ej: Windows, Linux	≅1000ms

2.1 Funcionamiento sobre máquina desnuda

En sistemas muy simples es habitual no usar un sistema operativo

- reduce coste y huella de memoria
- pero obliga a escribir software que proporcionaría el sistema operativo:
  - acceso a entrada/salida estándar (gráficos, red, USB, ...)
  - gestión de memoria secundaria
  - gestión del tiempo
  - gestión de la concurrencia
    - planificación
    - sincronización
  - gestión de la memoria

Un método sencillo y habitual para gestionar la concurrencia sobre máquina desnuda es el ejecutivo cíclico

La máquina desnuda

Servicios asociados a la gestión de la máquina

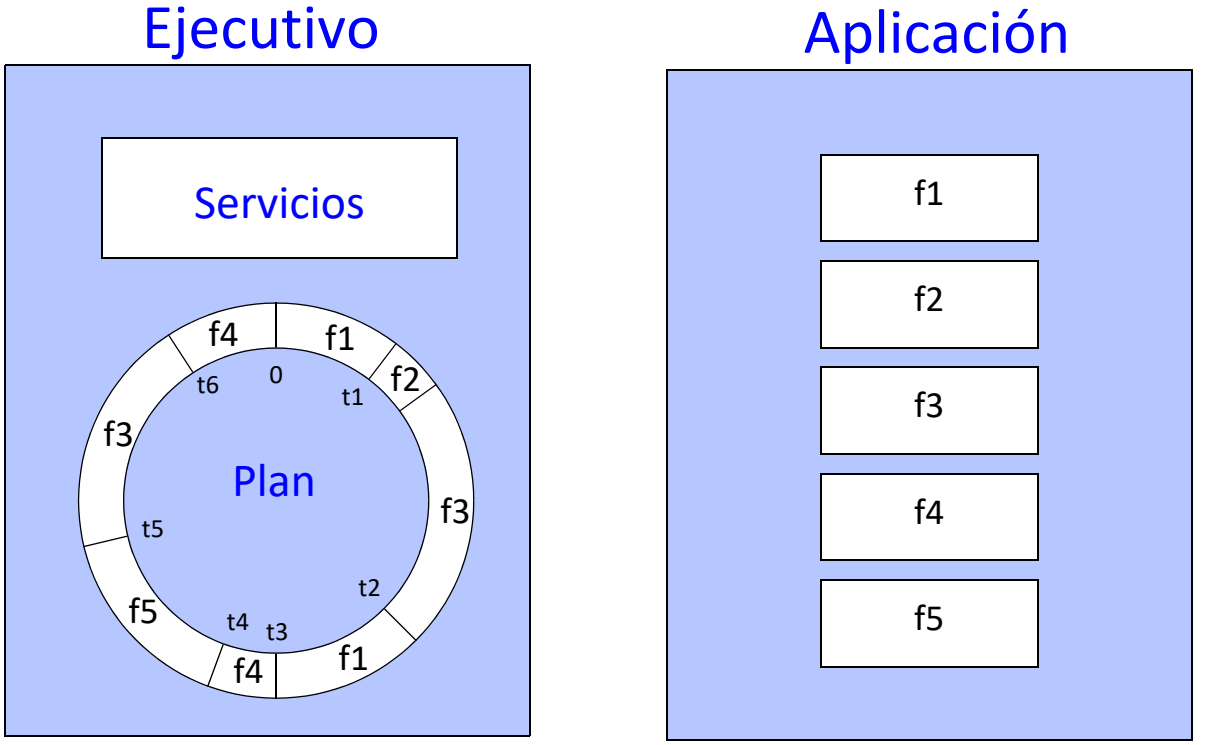
- entrada salida básica por puertos de I/O o mapeada en memoria
- gestión de interrupciones hardware y software
- gestión de mapas de memoria
- temporización

Sobre estos servicios de muy bajo nivel es preciso montar el software

Ejecutivo cíclico

Software usualmente dividido en dos partes:

- ejecutivo
  - funciona en modo supervisor (protegido)
  - atiende interrupciones hardware o software y del temporizador
- funciones de la aplicación
  - conjunto de funciones ejecutadas en modo usuario



Ejecutivos cíclicos

Reparten el trabajo entre las funciones de la aplicación por medio de un plan cíclico

- el plan detalla cuándo se va a invocar cada función del usuario
- el temporizador del sistema se programa para que envíe una interrupción
  - *periódica*: en ese caso los tiempos detallados en el plan deben ser múltiplos exactos del periodo
  - *programada*: en ese caso basta con que los tiempos del plan sean múltiplos de la resolución del reloj del sistema
- el tiempo ocioso se puede dedicar a ejecutar una función de segundo plano
  - contendría las acciones sin requisitos temporales
- es posible detectar incumplimientos en el plan, pero es difícil recuperarse de ellos

Ejemplo de ejecutivo cíclico

Actividades en un sistema simplificado de control de un robot:

Control de Servos

C=1 ms.
T=5 ms.
D=1.5 ms.

Planificación de Trayectorias

C=9 ms.
T=50 ms.
D=30 ms.

Reportero

C=73 ms.
T=1000 ms.
D=1000 ms.

C = Tiempo de ejecución (WCET)
T = Periodo
D = Plazo máximo de finalización

Notas:

El ejemplo anterior es una simplificación del sistema de control de un robot. Dispone de tres actividades que se deben ejecutar a la vez:

- Control de los servomotores
- Planificación de las trayectorias
- Informes periódicos del estado del robot en pantalla (reportero)

Junto a cada actividad se detallan sus requisitos temporales

- ***C***: Es el tiempo de ejecución de peor caso (WCET) que se tarda en ejecutar la actividad una sola vez, suponiendo que está ella sola en la CPU
- ***T***: Es el periodo, o intervalo en el que es necesario repetir la actividad. En muchos sistemas de control hay actividades periódicas, que hay que repetir constantemente
- ***D***: Es el plazo de ejecución (deadline). Es el intervalo máximo que puede transcurrir entre el comienzo del periodo y la finalización de la actividad. Es habitual que una tarea periódica tenga un plazo igual al periodo, lo que significa que la actividad se haya completado antes de comenzar el siguiente periodo. Esto pasa aquí con el reportero. Sin embargo, hay ocasiones en que una actividad es más urgente, y tiene un plazo menor al periodo. Por ejemplo el control de servos, requiere en este caso ser completado dentro de los primeros 1.5 milisegundos.

Plan cíclico

Es posible dar una solución no concurrente al programa anterior, intercalando la ejecución de las diferentes actividades

Sin embargo, podemos ver que el reportero ejecuta durante 73 ms

- no daría tiempo de ejecutar las otras actividades dentro de sus periodos

Solución:

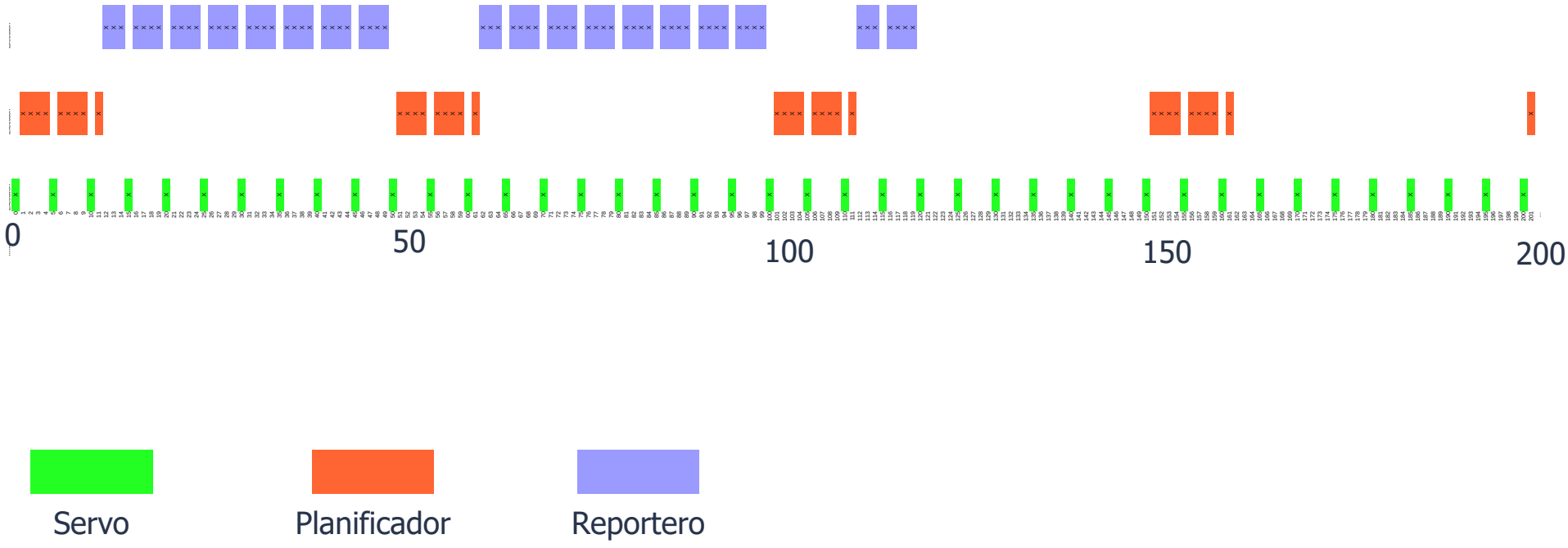
- partir las actividades largas en varios trozos
- hacer un plan cíclico, que se repita

Se construye un plan que se repite cíclicamente, con el mínimo común múltiplo de los periodos: 1000 en este caso

Programación no concurrente: ejecutivo cíclico

Al hacer el plan puede ser necesario partir funciones en varios fragmentos

El plan dura 1000ms, pero se muestra solo hasta el instante 200



Notas:

En la solución cíclica hemos hecho un plan que se repite cada 1000 ms

Hemos partido el Planificador de trayectorias en tres partes de tiempo de ejecución determinado: 4, 4 y 1ms

- Esta partición puede ser muy compleja. Si el código tiene instrucciones condicionales, bucles, etc., es muy difícil decidir por dónde cortamos para que el tiempo de ejecución sea el deseado.

¡Habrá que partir el Reportero en 24 partes!

Intercalando las ejecuciones como en la figura superior conseguimos cumplir todos los requisitos temporales (plazos y periodos).

Quedan bastantes intervalos ociosos en los que podemos aprovechar ese hueco para hacer otras cosas (de pequeña duración, cada una).

A continuación mostramos cómo sería la implementación de un ejecutivo cíclico que implementa esta solución.

- La variable `tiempo` toma valores en unidades de milisegundos, y nos permite saber en qué parte del plan estamos.

Programación no concurrente: ejecutivo cíclico

Requiere un sistema de ejecución muy sencillo

- una interrupción periódica
- una tabla de funciones (tareas) a invocar

Tarea	WCET	Iniciada en	Plazo
Control_De_Servos	1	0, 5, 10, 15, ...	1.5
Planificación_Trayectorias_1	4	1, 51, 101, ...	30
Planificación_Trayectorias_2	4	6, 56, 106, ...	30
Planificación_Trayectorias_3	1	11, 61, 111, ...	30
Reportero_1	3	12	1000
Reportero_2	4	16	1000
...así hasta 24 fragmentos			

Programación con ejecutivo cíclico

```
función Ejecutivo_Cíclico
   Entero Tiempo=0; -- Aumenta cada 5ms e indica el lugar en el plan
begin
   loop
      Esperar_Interrupcion; --Una cada 5 ms
      Control_De_Servos;
      if Tiempo=0 then
         Planificación_Trayectorias_1;
      elsif Tiempo=5 then
         Planificación_Trayectorias_2;
      elsif Tiempo=10 then
         Planificación_Trayectorias_3;
         Reportero_1;
      elsif Tiempo=15 then
         Reportero_2;
      elsif Tiempo=20 then
         ...
      end if;
      Tiempo:=Tiempo+5;
      if Tiempo=1000 then Tiempo=0; end if;
   end loop;
end Ejecutivo_Cíclico;
```

Programación con ejecutivo cíclico

El programa generado depende del plan cíclico diseñado:

- para un plan diferente habría que trocear el código de otra manera además de modificar el propio programa que ejecuta el plan
- cualquier cambio en el programa implica una gran dificultad para rediseñar los nuevos fragmentos y el nuevo plan

Conclusión:

- el ejecutivo cíclico es simple y barato, pero poco flexible y rudimentario

2.2 Sistemas operativos dirigidos por eventos

El uso de un sistema operativo es casi imprescindible en sistemas complejos, pues evita "reinventar la rueda", al proporcionar:

- gestión de procesos o hilos (threads) y de su planificación
- sincronización y paso de mensajes
- temporización
- gestión de memoria
  - protección de espacios de direcciones diferenciados, memoria virtual y memoria compartida
- almacenamiento masivo
- comunicaciones por red
- entrada/salida en dispositivos estándar:
  - pantalla, teclado, USB, RS-232

Gestión de dispositivos de entrada/ salida

Casi todos los sistemas empotrados requieren el uso de dispositivos de entrada/salida no estándar

- su programación se hace por el propio desarrollador de la aplicación, o terceras partes

El sistema operativo proporciona un marco estable para la instalación de los controladores de dispositivos

- basado en funciones estándar como `open`/`close`/`read` y `write`, así como un comodín llamado `ioctl`
- este marco facilita la migración de las aplicaciones, pero no de los propios controladores que deben cambiar al migrar de un hardware a otro

Sistemas dirigidos por eventos

Tiempo real en un sistema operativo (POSIX):

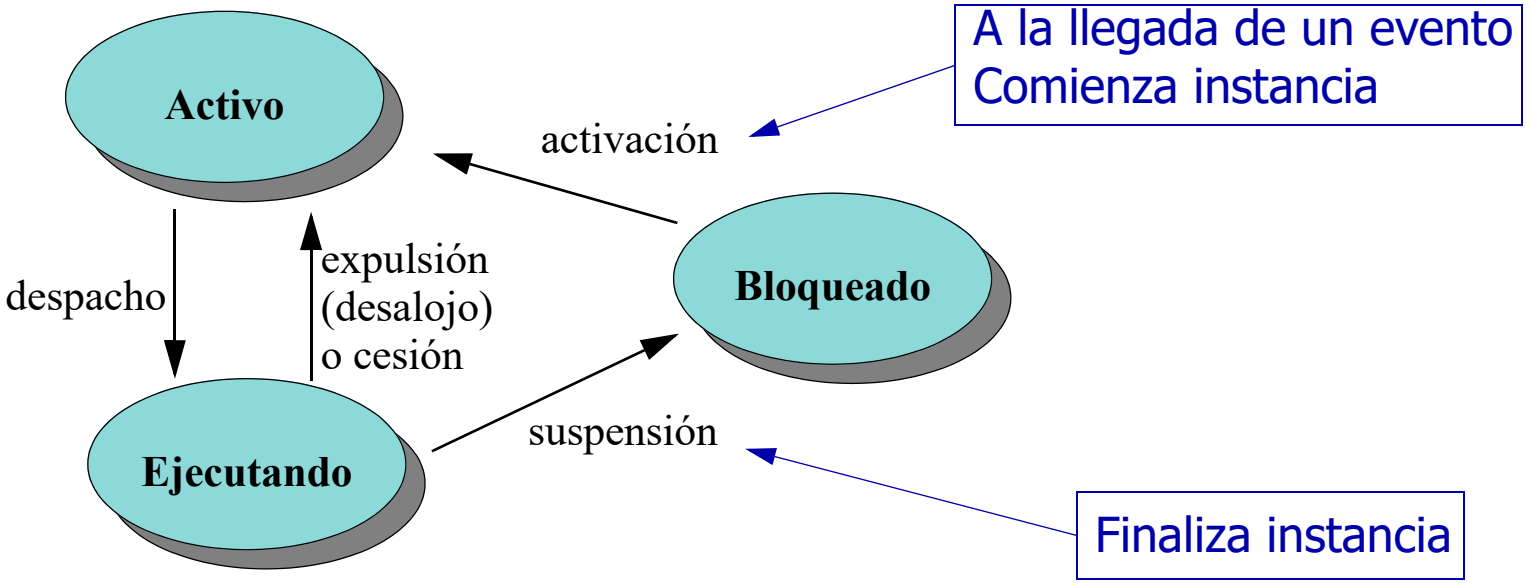
- "La habilidad del sistema operativo para proporcionar el nivel de servicio requerido con un tiempo de respuesta acotado"

La aplicación se organiza como un conjunto de procesos o threads que se ejecutan en concurrencia

- *proceso*: objeto que contiene uno o varios hilos de ejecución y un espacio de direcciones único
- *thread*: hilo de ejecución que se ejecuta en concurrencia con los demás hilos

Planificación de threads

Cada thread o actividad concurrente se programa de manera independiente y puede estar en uno de estos estados:



El planificador elige de entre los hilos activos uno para ejecutar por cada CPU

- la elección es por la prioridad, según una política de planificación

Políticas de planificación

Según la capacidad de desalojo:

- *expulsora*: cuando se activa un thread de alta prioridad se desaloja de la CPU a un thread en ejecución
- *no expulsora*: un thread que está ejecutando solo pasa al estado activo voluntariamente (al terminarse o por *cesión*)
  - también se llama planificación cooperativa

Según la gestión de las prioridades

- *prioridades fijas (FP)*: solo cambian explícitamente
- *prioridades dinámicas*: cambian según avanza el tiempo o las condiciones del sistema
  - las más habituales son EDF (Earliest Deadline First) y Round Robin

Existen variantes compatibles con ellas para tratar eventos aperiódicos

Política FP

Veremos ahora la planificación expulsora por prioridades fijas en POSIX

- a cada tarea se le asigna una *prioridad base*, que no cambia salvo que se haga explícitamente
- las tareas pueden tener una *prioridad activa* que temporalmente puede ser más alta que la prioridad base
  - la prioridad activa es la máxima entre la prioridad base y todas las prioridades *heredadas*
  - una tarea puede heredar prioridades a causa de la sincronización

La prioridad activa es la utilizada por el planificador

Si la planificación es desalojante (expulsora):

- cuando una tarea de alta prioridad se activa, desaloja a otra tarea de menor prioridad que se esté ejecutando

Ejemplo con prioridades fijas: sistema de control del robot

main

```
Set_Priority (self,20)
th1=crea_tread (Th_Control_De_Servos);
th2=crea_tread (Th_Planificador_Trayectorias);
th3=crea_tread (Th_Reportero);
th4=crea_tread (Th_Background);
Set_Priority (th1,10);
Set_Priority (th2,8);
Set_Priority (th3,6);
Set_Priority (th4,1);
```

Ejemplo: sistema de control del robot

Th_Control_De_Servos

```
loop
   Control_De_Servos;
   next:=next+0.005;
   Sleep_Until next;
end loop;
```

Th_Planificador_Trayectorias

```
loop
   Planif_Trayectorias;
   next:=next+0.05;
   Sleep_Until next;
end loop;
```

Th_Reportero

```
loop
   Reportero;
   next:=next+1.0;
   Sleep_Until next;
end loop;
```

Th_Background

```
loop
   hacer cosas
     en 2º plano;
   ...
end loop;
```

Ejercicio 2.1

Ejercicio 2.1. Analizar la planificabilidad del sistema de control de robots mediante MAST

- Suponer que la tarea de 2º plano tiene un tiempo de ejecución promedio de 2 segundos, sin requisitos temporales
  - Para el análisis haremos una estimación razonable de su tiempo de ejecución de peor caso
- ¿Aproximadamente, qué periodo mínimo podemos ponerle para que se pueda ejecutar con una holgura razonable?

Prioridades dinámicas

La prioridad puede variar según el paso del tiempo o el estado del sistema

- es más flexible, lo que permite un uso más eficiente de los recursos

Dos tipos

- prioridades de instancia estáticas:
  - cada instancia de una tarea tiene una prioridad estática
  - la prioridad puede variar de una instancia a la siguiente
  - Ej.: *EDF* (Earliest Deadline First); cada instancia tiene como prioridad su plazo absoluto
- prioridades de instancia dinámicas:
  - la prioridad puede cambiar en mitad de la ejecución de la instancia
  - Ej.: *LLF* (Least Laxity First); la prioridad de la instancia es el inverso de su laxitud
    - Laxitud: el plazo absoluto menos el tiempo de ejecución restante

Prioridades dinámicas

En general, las prioridades de instancia dinámicas:

- son más complejas
- tienen más sobrecarga de ejecución

Las prioridades de instancia estáticas:

- son más utilizadas
- son más simples
- *EDF* y sus variantes son las más frecuentes

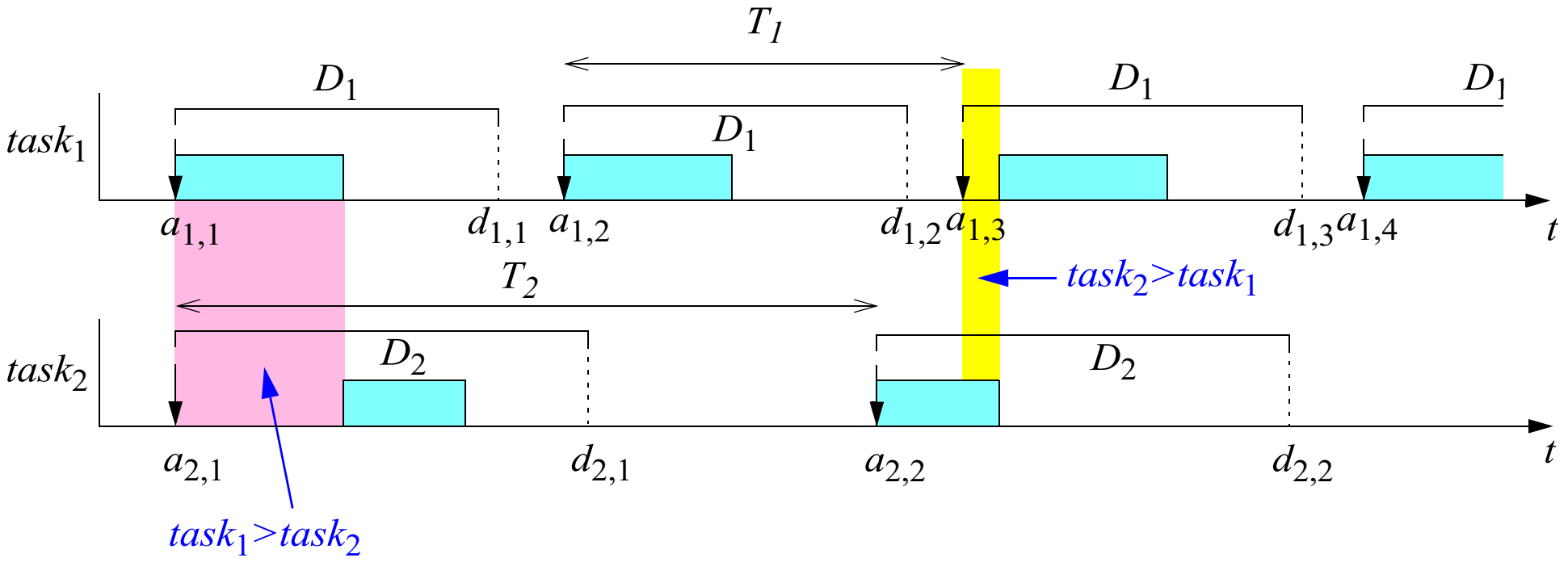
Política EDF

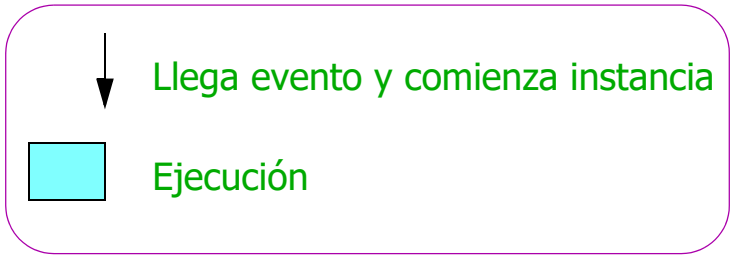
La teoría de planificabilidad muestra que en general es posible conseguir un mayor rendimiento usando prioridades dinámicas, en lugar de fijas

Uno de los algoritmos más estudiados es el ***EDF*** (Earliest Deadline First)

- 100% de utilización con tareas periódicas independientes en monoprocesador
- Cada tarea i tiene un periodo T_i y un plazo relativo D_i
- Cada instancia j de la tarea i tiene un instante de activación $a_{ij} = a_{i0} + jT_i$ y un plazo absoluto $d_{ij} = a_{ij} + D_i$
- La prioridad es el inverso del plazo absoluto
  - Se ejecuta primero la tarea con el plazo absoluto más próximo
- A cada instancia de la tarea cambia su plazo absoluto y por tanto la prioridad

Política EDF





Ejercicio 2.2

Ejercicio 2.2. Analizar con MAST la planificabilidad del sistema de control de robots con planificación EDF

- Suponer que la tarea de 2º plano tiene un tiempo de ejecución promedio de 2 segundos, sin requisitos temporales
- ¿Aproximadamente, qué periodo mínimo podemos ponerle para que se pueda ejecutar con una holgura razonable?
- ¿Qué diferencias has encontrado entre este sistema planificado con FP y EDF?

Política LLF (Least Laxity First)

El planificador elige para ejecución a la tarea con menor laxitud (laxity)

- laxitud: el intervalo de tiempo hasta el plazo absoluto menos el tiempo de cómputo restante hasta completar la instancia de la tarea

$$s_{ij} = d_{ij} - t - rc_{ij}$$

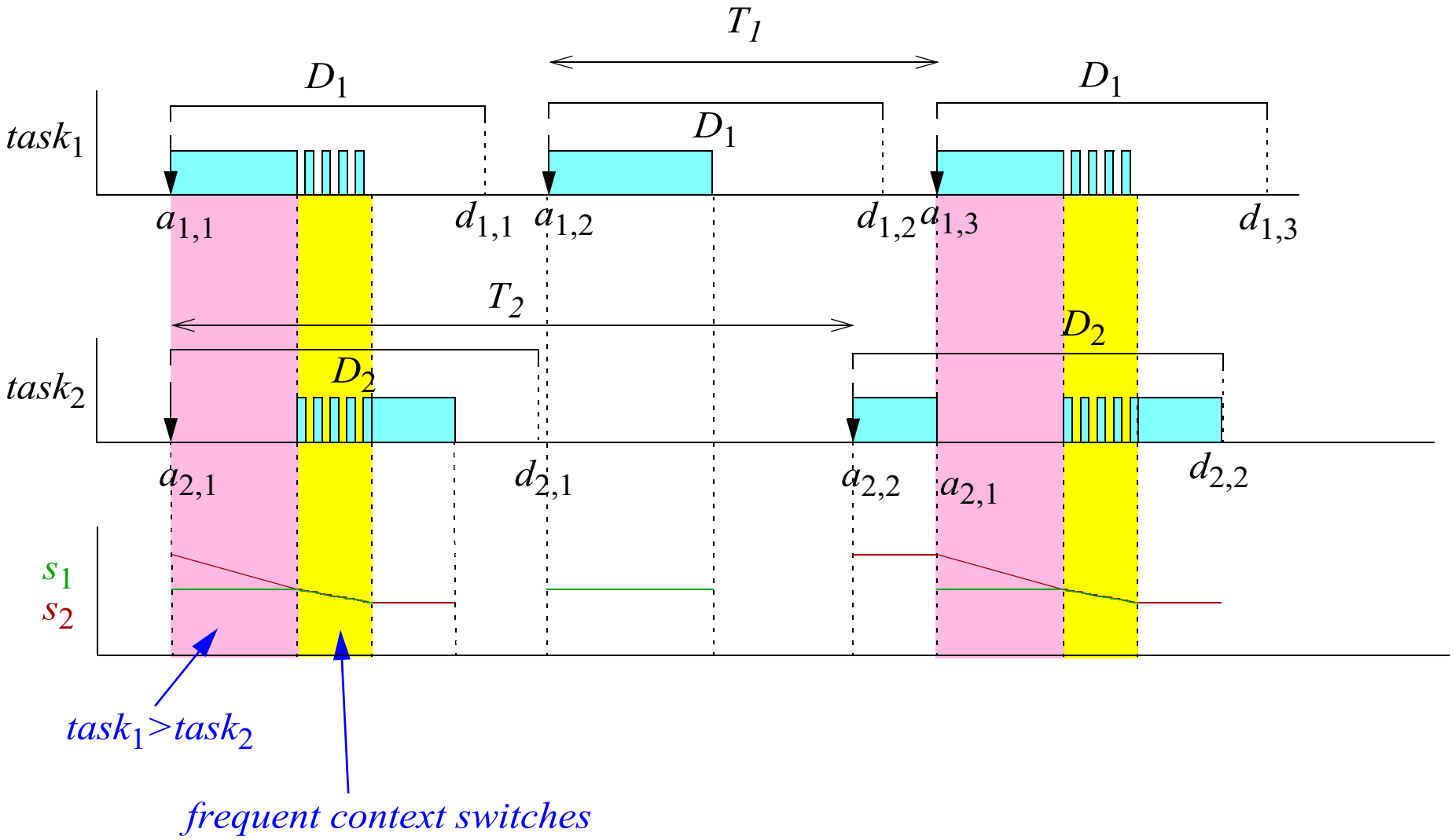
siendo s_{ij} la laxitud (holgura), rc_{ij} el tiempo de cómputo restante y t el tiempo absoluto

Es un ejemplo de un algoritmo en el que la prioridad de la instancia va cambiando dinámicamente

Resulta en frecuentes cambios de contexto y es poco práctica

Hay modificaciones a esta política con menos cambios de contexto

Política LLF



Política Round-Robin

Los hilos se turnan cíclicamente en el uso de la CPU

- Cuando pasa un *cuanto* de tiempo se cede la CPU al siguiente hilo

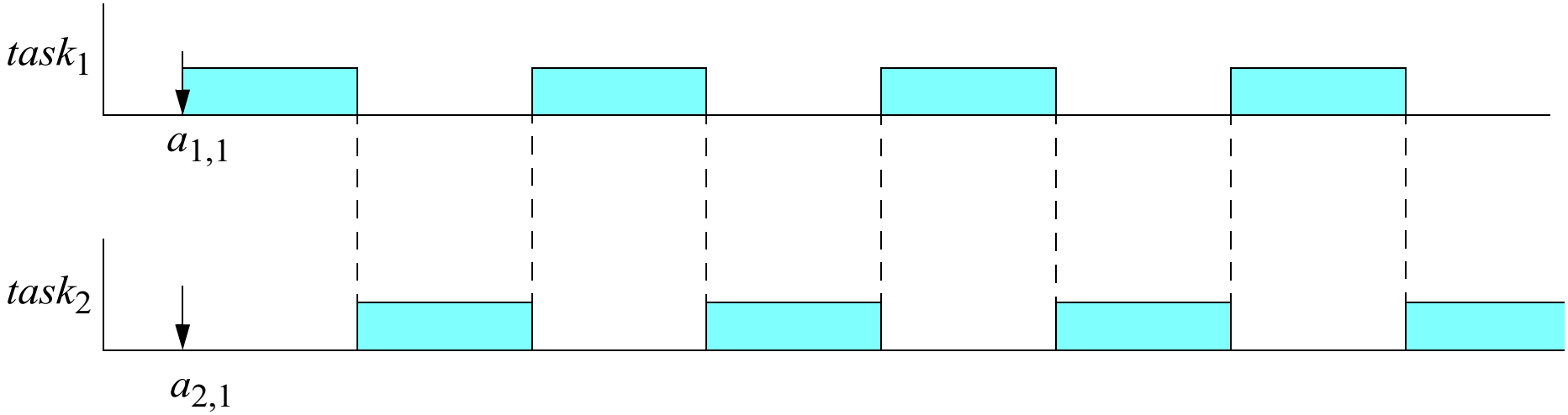
No hay garantías temporales si el número de tareas es variable

Es una política adecuada para tareas *sin* requisitos de tiempo real

En sistemas de tiempo real a menudo se usa de forma jerárquica con prioridades fijas

- situada en el nivel de prioridad más bajo
- donde se colocan los threads sin requisitos temporales

Política Round Robin



La inversión de prioridad

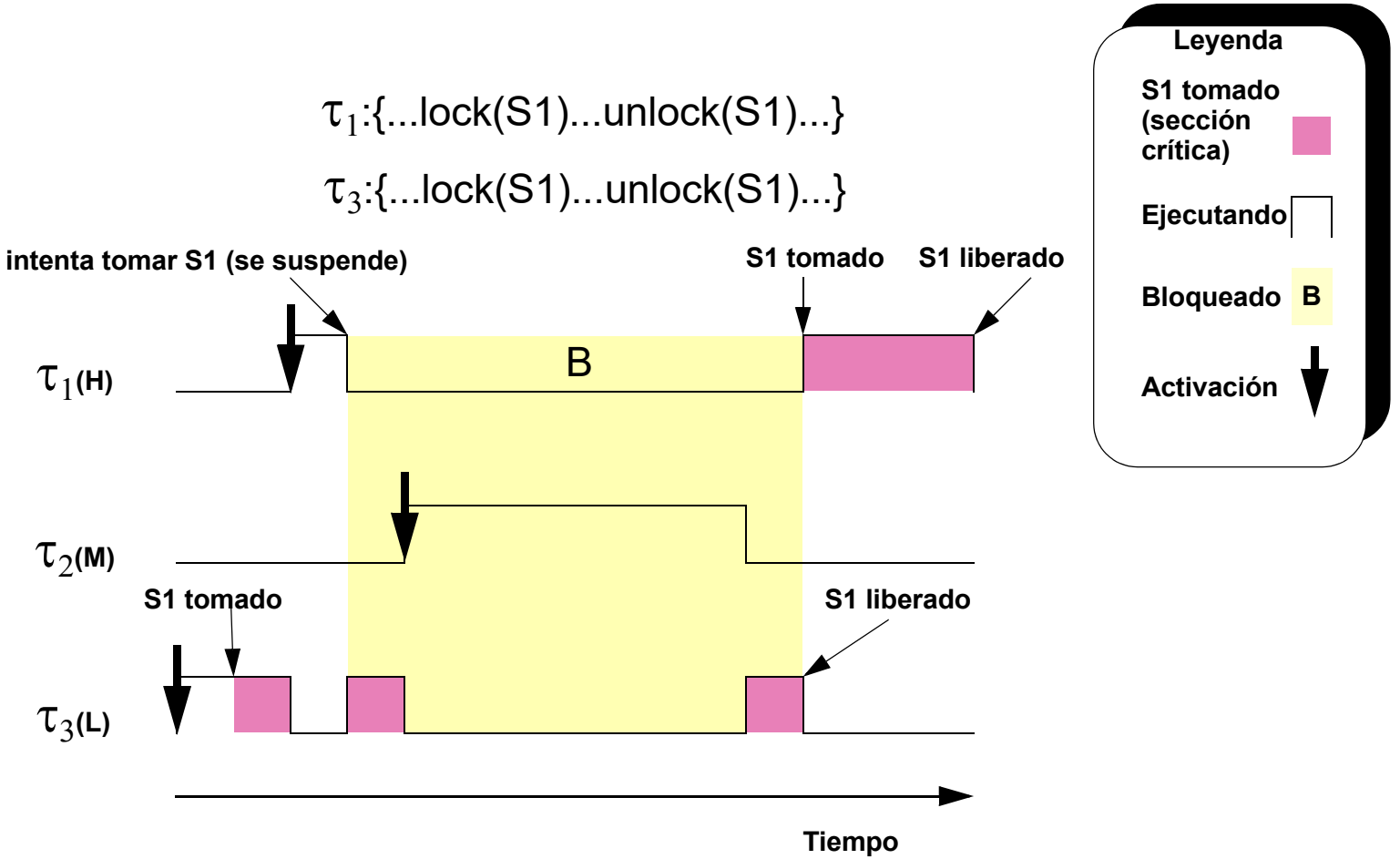
En sistemas de tiempo real es preciso evitar o minimizar situaciones de inversión de prioridad:

- en que una tarea de alta prioridad espera a que una de baja prioridad termine una acción

Fuentes de inversión de prioridad

- colas FIFO en acceso al hardware, colas de mensajes o eventos y servicios del sistema operativo
  - se soluciona usando colas de prioridad
- exclusión mutua para el uso de recursos compartidos
  - es aceptable una pequeña cantidad de inversión de prioridad si el acceso al recurso es de breve duración
  - pero es necesario evitar la inversión de prioridad *no acotada*
  - se soluciona usando protocolos de sincronización de tiempo real

Inversión de prioridad no acotada



Ocurre cuando una sección crítica es expulsada por una tarea de prioridad intermedia

Protocolos de sincronización de tiempo real

Evitan la expulsión durante la sección crítica por tareas que podrían producir inversión de prioridad no acotada

Protocolos para prioridades fijas

- no expulsión
- herencia de prioridad (BIP, *Basic Inheritance Protocol*)
- techo de prioridad inmediato (IPC, *Immediate Priority Ceiling*)
  - requiere especificar un techo de prioridad para cada recurso: debe ser la máxima prioridad de las tareas que lo usan

Protocolos para prioridades dinámicas (EDF)

- herencia de prioridad dinámica
- SRP (*Stack Resource Protocol*)

Comparación de protocolos FP

Protocolo	No expulsión	BIP	IPC
Require precalcular techos de prioridad	No	No	Sí
Bloqueo por una sola sección crítica	Sí[2]	No	Sí[2]
Evita deadlocks	Sí[1]	No	Sí[1]
Evita bloqueo innecesario	No	Sí	Sí
Tiempo de respuesta promedio	Igual al peor caso	Bajo	Igual al peor caso
Sobrecarga por cambios de contexto	0	$2C_s$ por sección crítica	0

[1] **Solo si las tareas no se suspenden en el interior de secciones críticas**
[2] **Solo si las tareas no se suspenden**

Sistemas operativos con planificación por prioridades

Estándares:

- POSIX:
  - Integrity, VxWorks, QNX, LynxOS, Nucleus RTOS, RTEMS, MaRTE OS
- ITRON y μTRON
  - eCOS, RTEMS

Otros sistemas operativos

- RODOS
- OnTime
- RTXC Quadros

2.3 Reserva de recursos

Solución intermedia entre los sistemas dirigidos por eventos y los completamente particionados, ideal para tareas *aperiódicas*

Consiste en utilizar técnicas de planificación que limiten el uso de un determinado recurso por parte de un componente

En el caso de la CPU o la red se limita la anchura de banda

- tiempo máximo de uso en una determinada ventana temporal
- asociado con un plazo temporal

En la red se pueden usar los mismos algoritmos que en las CPUs

- el tiempo de ejecución se corresponde con el tiempo de envío de mensaje
  - relacionado con la longitud del mensaje y la velocidad de la red
- los paquetes que componen un mensaje no son expulsables

Algoritmos más habituales de reserva de recursos

Servidor periódico (polling server)

- una tarea periódica ejecuta el componente cuando detecta la llegada de un evento
- el tiempo de ejecución de esta tarea está limitado por el sistema operativo
  - si no da tiempo de ejecutar la respuesta al evento en un periodo se sigue en el siguiente, con el consiguiente retraso
- no hay relación entre la llegada de eventos y el periodo del servidor
  - ello supone retrasos de hasta un periodo

Algoritmos de reserva de recursos (cont)

Servidor aplazable (deferrable server)

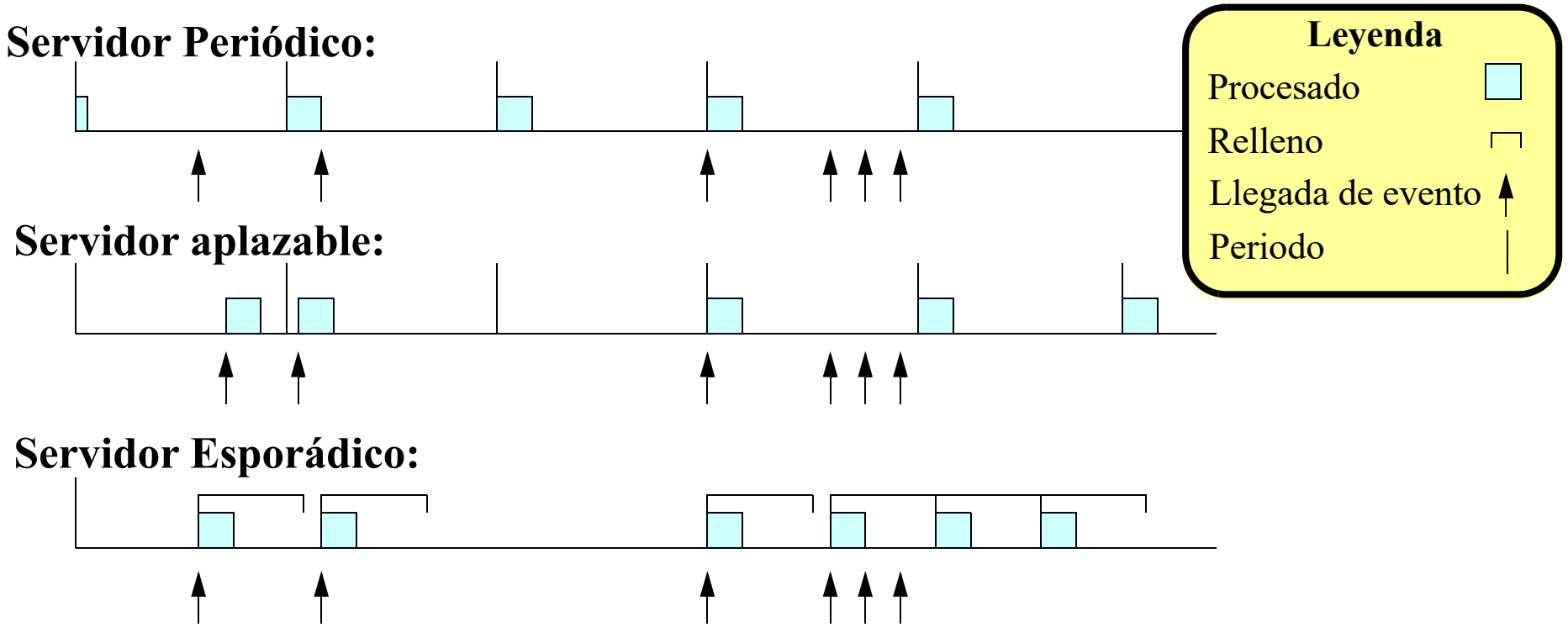
- el servidor se ejecuta cuando detecta la llegada de un evento
  - no hay retraso
- el servidor tiene limitada la capacidad de ejecución
  - cuando se gasta la capacidad se detiene
  - o baja la prioridad a un nivel muy bajo, de segundo plano
- periódicamente se rellena la capacidad del servidor a su valor inicial
  - el periodo de relleno no tiene relación con la llegada de eventos
  - pueden acumularse dos ejecuciones próximas si un evento llega tarde y justo después de ejecutarse se hace un relleno y llega un nuevo evento
  - esto produce mayor sobrecarga sobre tareas de prioridad inferior

Algoritmos de reserva de recursos (cont)

Servidor esporádico (sporadic server)

- el servidor se ejecuta cuando detecta la llegada de un evento
  - no hay retraso
- el servidor tiene limitada la capacidad de ejecución
  - cuando se gasta la capacidad se detiene
  - o baja la prioridad a un nivel muy bajo, de segundo plano
- por cada porción de capacidad gastada ésta se rellena en la misma cantidad un intervalo después de haberse activado su ejecución
  - el intervalo se llama el periodo de relleno
  - la ejecución se activa cuando llega el evento y hay capacidad para ejecutarlo (lo que ocurra más tarde)
- es el algoritmo más complejo, pero el que tiene mejores prestaciones:
  - bajo tiempo de respuesta
  - baja sobrecarga sobre tareas de prioridad inferior

Comparación de algoritmos de reserva de recursos



En los tres casos el servidor tiene dos parámetros:

- capacidad inicial y periodo (de relleno)

Ejemplo: Arquitectura dirigida por eventos para el sistema SCADA

Ejercicio 2.3. Plantear una arquitectura dirigida por eventos para el ejemplo SCADA

- Explorar con MAST la planificabilidad, protegiendo la temporización de la tarea "check" con un servidor esporádico con:
  - una capacidad inicial superior al WCET
  - y un periodo igual o inferior al mínimo tiempo entre llegadas
- Hallar los parámetros del servidor esporádico que hacen el sistema planificable con una holgura razonable
  - jugar con la capacidad inicial

2.4 Particionado en el espacio y el tiempo

Para gestionar la complejidad de los sistemas se tiende a separar subsistemas en distintos componentes

- quizás desarrollados por diferentes contratistas

Para garantizar su independencia se solían desarrollar en CPUs dedicadas (arquitectura *federada*)

- coste alto en cantidad de hardware, comunicaciones y cableado

Hoy se tiende a soluciones en las que se integran varios componentes en la misma CPU (arquitectura *modular integrada*, IMA)

- esto permite tener sistemas de criticidad mixta

Para garantizar su independencia:

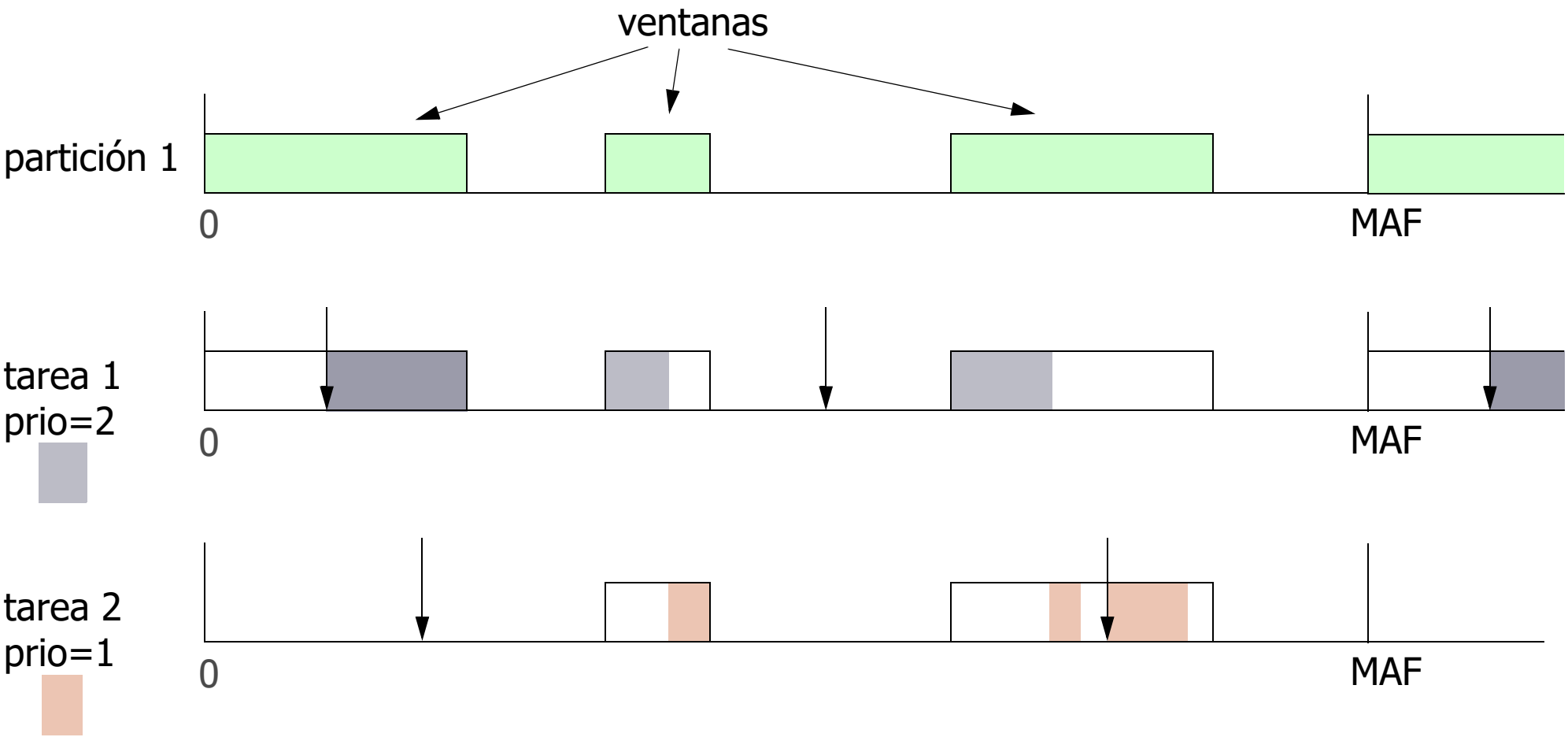
- particionado espacial (memoria, I/O)
- particionado temporal (CPU, comunicaciones)

Estándares para particionado

ARINC-653: Estándar en el dominio de la aeronáutica

- Define una API (Application Program Interface) de un sistema operativo particionado llamada APEX
- Se pueden definir particiones compuestas cada una de ellas por un conjunto de una o varias ventanas temporales
  - de comienzo y duración definibles
  - protegidas en tiempo y espacio
- El plan de ejecución de las ventanas se repite cíclicamente en un intervalo llamado el MAF (*Major Frame*)
- Cuando varias tareas comparten la misma partición se puede usar un planificador secundario, por ejemplo de prioridades fijas
- La I/O (entrada/salida) se hace por tareas que hacen muestreo periódico en particiones especiales de I/O
- La partición permite la certificación independiente de cada una

Ejemplo de planificación ARINC-653



Las tareas 1 y 2 están asignadas a la partición 1
La partición 1 tiene un planificador secundario de prioridades fijas (T1>T2)

Comunicación inter-partición en ARINC-653

La comunicación es por puertos de dos tipos:

- *muestreado* (*sampling*): Se guarda un único dato que se modifica al sobreescribirlo y se puede leer múltiples veces
- *con cola* (*queuing*): Hay una cola de mensajes que se consumen al leerse

Se pueden crear "*end-to-end flows*" mediante puertos *con cola*, en los que es posible esperar a la llegada de un mensaje

Los puertos *muestreados* se usan más bien para desacoplar la ejecución de actividades con periodos diferentes y que comparten datos

Implementaciones ARINC-653

Selección de las más usuales:

- VxWorks 653 (Wind River)
- Integrity 178B (Green Hills)
- PikeOS (Sysgo)
- LynuxWorksOS-178 (Lynx)
- LithOS/Xtratum (fentISS)

Ejercicio

Ejercicio 2.4. Explorar la planificabilidad del sistema SCADA con la arquitectura particionada en el tiempo tipo ARINC-653 sobre plataforma monoprocesadora

- No hay rutinas de servicio de interrupción
- El acceso al hardware se hace siempre desde la partición de entrada/ salida
- No hay recursos compartidos
  - Se usan los puertos *queueing* o *sampling* para la comunicación entre tareas
- Los eventos aperiódicos se tratan mediante la técnica del muestreo periódico

Estándares para particionado (cont.)

AUTOSAR (AUTomotive Open System ARchitecture)

- Estándar en el dominio de la automoción
- Permite especificar componentes que implementan funciones complejas, potencialmente distribuidas
  - compuestos por *componentes software* más pequeños, que no son distribuidos
- Reutilización de software
- Flexibilidad de diseño
- Facilidad de integración
- Competitividad, al permitir componentes de diferentes fabricantes
- Tests estandarizados

Socios principales

BMW Group

Bosch

Continental

Daimler

Ford

General Motors

PSA Group

Toyota

Volkswagen Group

Fuente: https://www.autosar.org/about/current-partners/core-partners/

Particionado en sistemas AUTOSAR

La integración se complica en sistemas complejos pues depende del comportamiento de todos los componentes

- *no hay aislamiento temporal*

La fiabilidad se puede ver comprometida por la protección de memoria

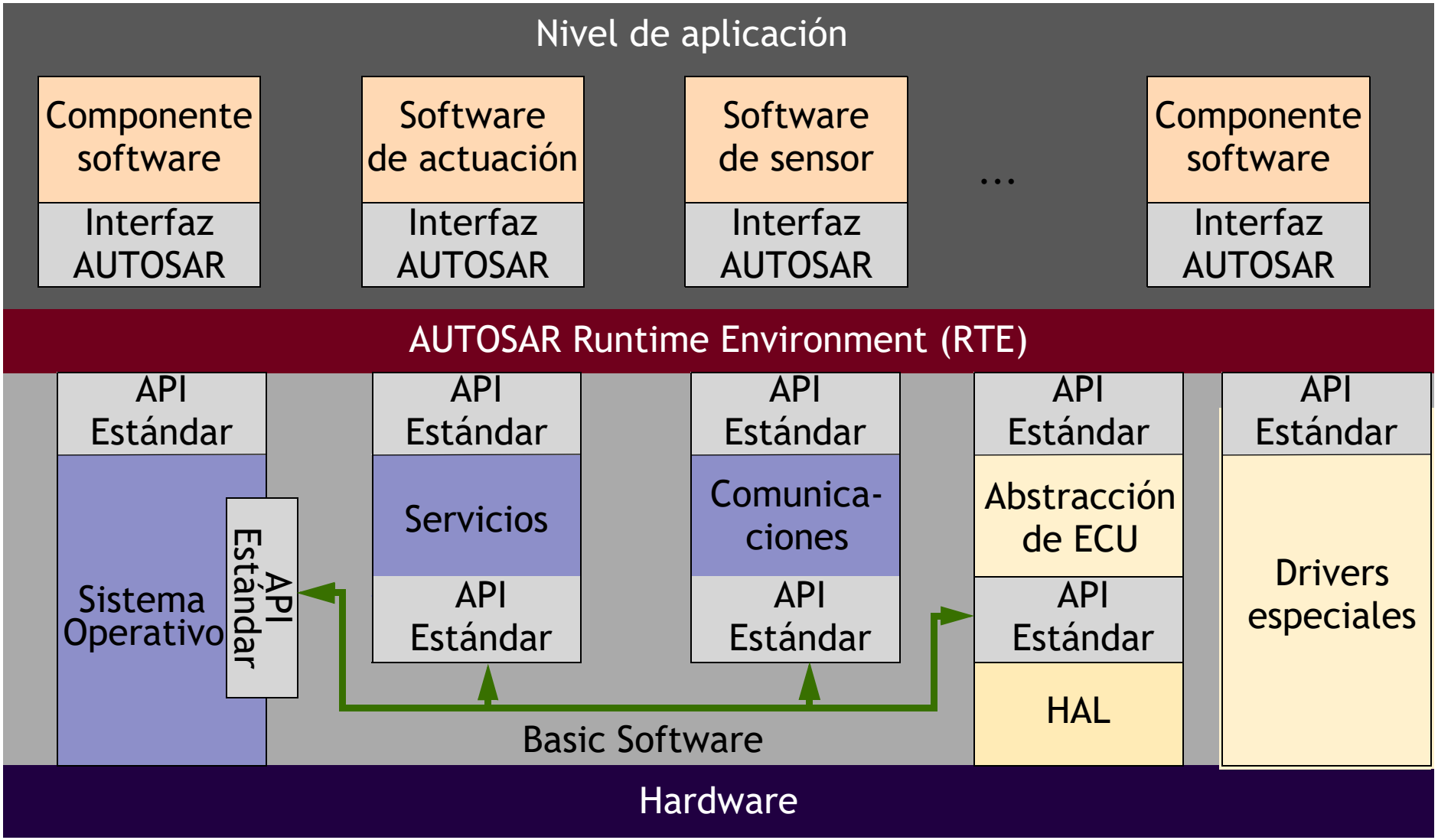
- según el hardware puede haber o no *protección de memoria*

Cuando se persigue la fiabilidad es preciso elegir implementaciones de AUTOSAR que:

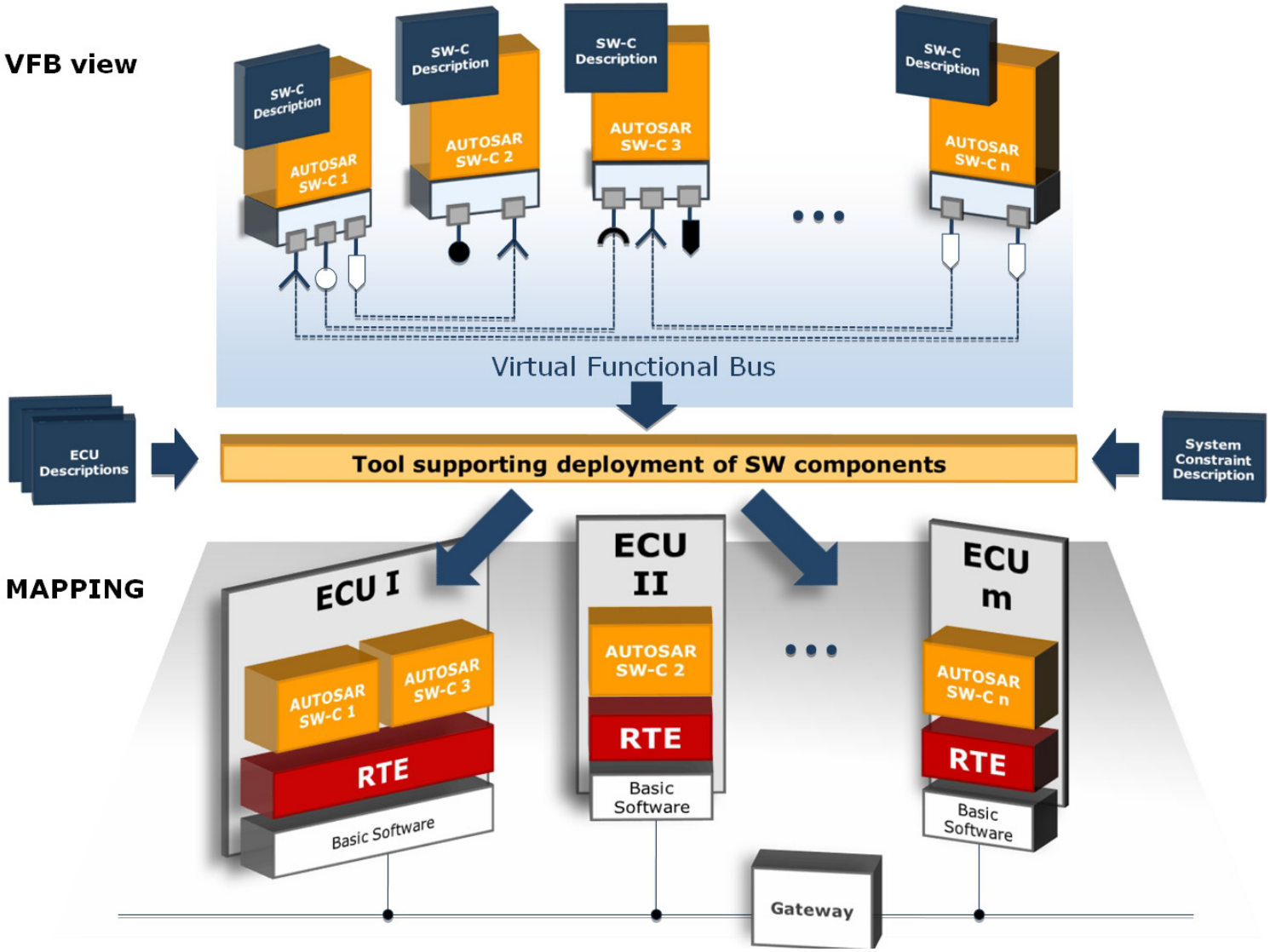
- Tengan funciones de protección de memoria por hardware
- Tengan funciones de protección temporal para asegurar que cada componente software verifica sus requisitos temporales

También es interesante realizar comprobaciones de la integridad de los mensajes y datos de I/O

Arquitectura AUTOSAR



Configuración en AUTOSAR



Implementaciones de AUTOSAR

Selección de vendedores que proporcionan un sistema operativo y herramientas AUTOSAR completos

- Elektrobit (ahora parte de Continental AG)
- embitel
- ETAS
- KPIT
- Mentor Graphics (ahora parte de Siemens)
- Vector Informatik

2.5 Particionamiento hardware-software

En los sistemas empotrados el hardware y el software no se pueden considerar independientemente:

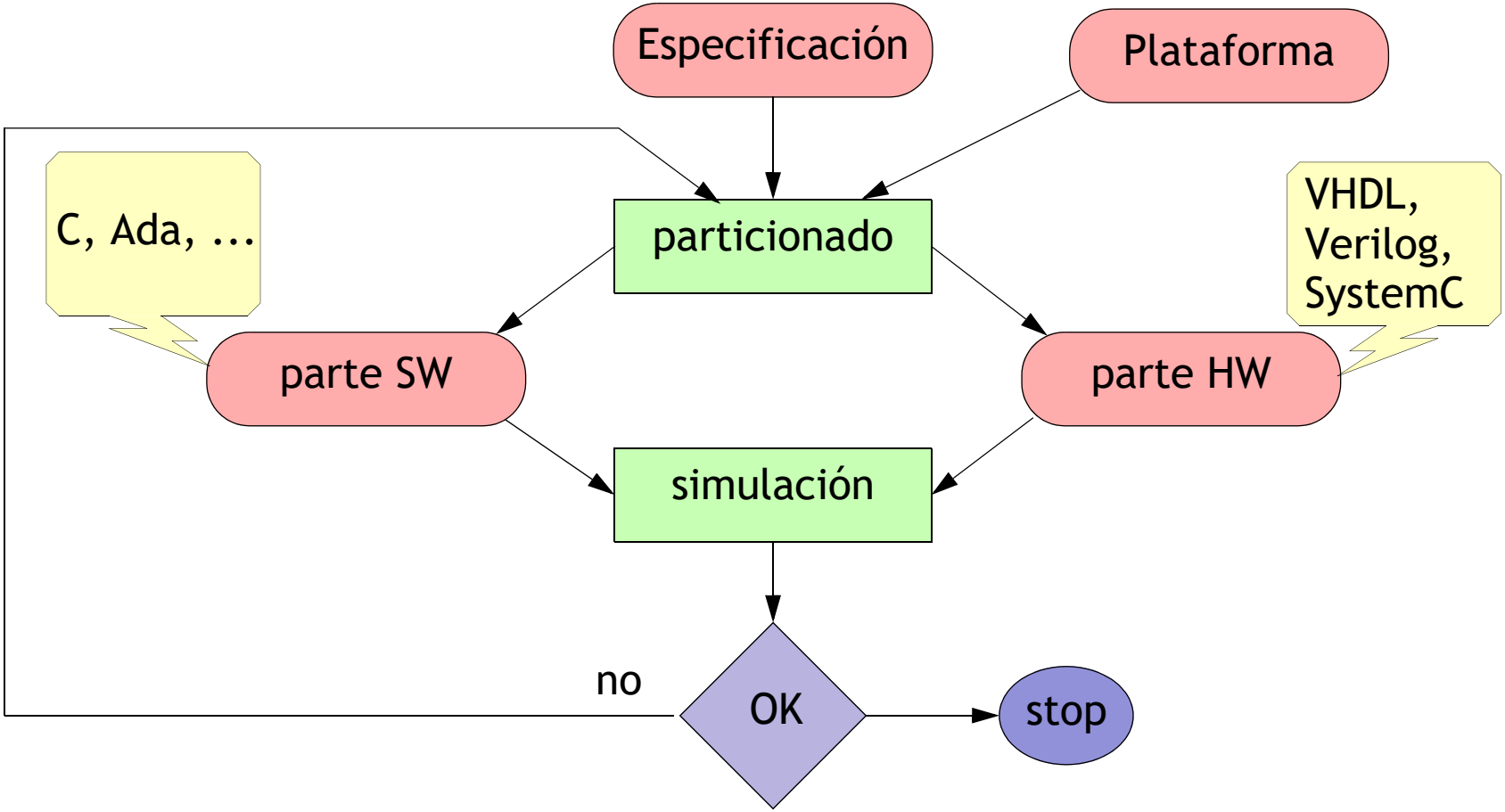
- codiseño hardware/software
- el objetivo es encontrar la mejor combinación de hardware y software que nos lleve al producto más eficiente capaz de cumplir la especificación

La mayor potencia de cálculo de los procesadores actuales permite mover más funcionalidad del hardware al software

El objetivo general de la reutilización nos lleva a contar con componentes ya disponibles

- esto nos lleva al diseño basado en plataformas

Proceso para el particionado HW/SW



Particionado

En la especificación se describe la funcionalidad mediante tareas y sus dependencias mediante un grafo en el que cada arco representa comunicación entre tareas

- los nudos del grafo tienen información sobre el tiempo de ejecución de la tarea y el coste de implementación
  - tanto si se hace en SW como en HW
- los arcos del grafo tienen información sobre el coste de las comunicaciones
  - tanto si se hace en SW o en HW y si es entre componentes diferentes o en el mismo componente

Mediante un algoritmo heurístico se busca la solución óptima

- algoritmos de biparticionado en grafos
- alg. de optimización: investigación operativa, programación entera

Herramientas y lenguajes

SpecC

- lenguaje de especificación de sistemas y metodología de diseño

IMEC

- especificación en UML, Java y C++ concurrente

COSYMA

- especificación de sistemas de control en C^x (superconjunto de C)

Ptolemy II

- mezcla muchas especificaciones, para construir sistemas híbridos

OCTOPUS

- métodos de especificación clásicos como diagramas de casos de uso, máquinas de estados y diagramas de clases

2.6 Plataformas basadas en lenguajes de programación concurrentes

Algunos lenguajes de programación especifican servicios de concurrencia

- pueden ofrecer una implementación sobre máquina desnuda que se puede usar como plataforma para sistemas empotrados

El ejemplo más interesante es Ada, que además tiene soporte opcional para tiempo real

Otro ejemplo con interés reciente es Java de tiempo real

- pero tiene pocas implementaciones y menos sobre máquina desnuda

Los lenguajes C y C++ especifican concurrencia a partir de sus estándares de 2011 (C-11 y C++-11)

- pero no especifican servicios de tiempo real

Ada

El lenguaje Ada soporta la programación de sistemas empotrados y de sistemas de tiempo real, mediante los siguientes mecanismos:

- Representación del hardware
- Interrupciones
- Gestión del tiempo
- Planificación por prioridades fijas y/o EDF
- Sincronización libre de inversión de prioridad

Estos mecanismos están descritos en anexos opcionales:

- Anexo de programación de sistemas
- Anexo de sistemas de tiempo real

Ada (cont.)

Ada utiliza planificación de tareas *expulsora* (o desalojante)

- Por prioridades *fijas*
  - con orden FIFO para la misma prioridad
  - o con orden cíclico para la misma prioridad (Round Robin)
- Por prioridades *dinámicas* (EDF)
- Permite mezclarlas en un sistema de planificación *jerárquica* a dos niveles

También define una política *no expulsora* por prioridades fijas

Java de tiempo real

Java es un lenguaje moderno que soporta la programación orientada a objetos, la programación concurrente, ...

Java tiene soporte opcional para sistemas de tiempo real

- RTSJ: Real-time specification for Java

Soporta threads con características de tiempo real

- planificación por prioridades fijas o EDF

Tiene una dificultad importante en la gestión de la memoria

- La memoria automática de Java no es muy adecuada para sistemas de tiempo real; RTSJ define otras formas de gestión de memoria

Ha existido mucho interés en RTSJ pero escasas aplicaciones e implementaciones, aún

2.7 Sistemas empotrados distribuidos

Muchos sistemas empotrados modernos son distribuidos

- Se distribuye la inteligencia para que esté *próxima* a los sensores y actuadores
- Se puede establecer *colaboración* entre sistemas o subsistemas
- Arquitecturas *federadas*:
  - cada subsistema tiene su propio computador y hay que unir todos

Vamos a analizar con un poco más de detalle dos redes concretas

- Bus CAN: red basada en prioridades fijas
- TTP: red basada en el particionado temporal

Analizaremos también las redes de sensores

Bus CAN

El bus CAN (Controller Area Network) es una red de campo ampliamente difundida en el mundo del automóvil y en la industria

Permite la comunicación entre computadores y actuadores sin necesidad de un árbitro central

Utiliza un protocolo basado en mensajes cortos (hasta 8 bytes), con prioridad

Se desarrolló en 1983 por Robert Bosch GmbH

- los primeros chips salieron al mercado en 1987 (Intel y Philips)

La última versión, CAN 2.0, se publicó en 1991 con dos partes

- CAN 2.0A: identificadores estándar de 11 bits
- CAN 2.0B: identificadores extendidos de 29 bits

Bus CAN (cont.)

CAN se estandarizó como ISO-11898

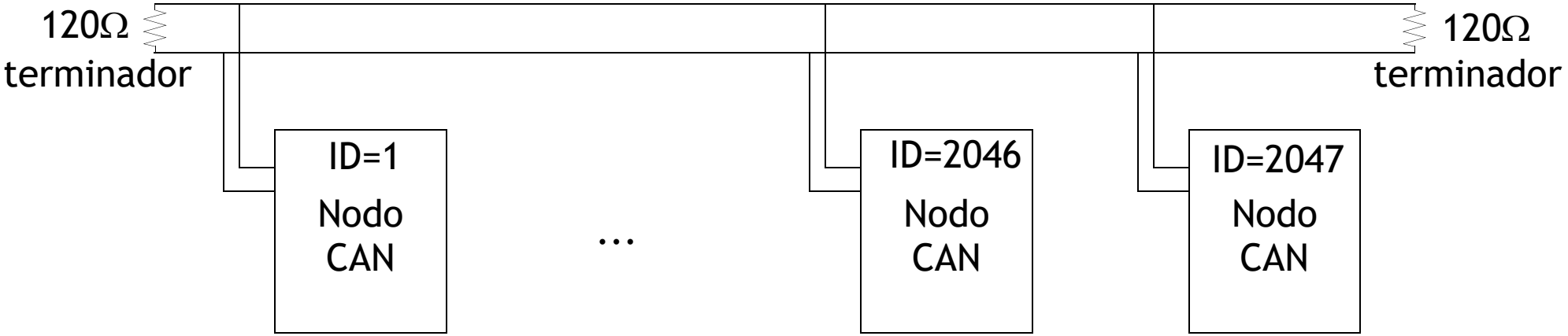
- Parte 1: nivel de datos
- Parte 2: nivel físico
  - 1Mbit/s hasta 40 m
  - 125Kbits/s hasta 500m
- Parte 3: nivel físico tolerante a fallos y de baja velocidad
- Parte 4: TTCAN o CAN disparado por tiempo
- Parte 5: nivel físico con modo de baja potencia
- Parte 6: nivel físico extensión del anterior con facilidades para despertar dispositivos de manera selectiva

Bus CAN (cont.)

Bosch siguió evolucionando la especificación y en 2012 sacó CAN FD 1.0 (CAN with Flexible Data-Rate)

- Usa un formato de frame diferente que permite cambiar la longitud (hasta 64 bytes) y velocidad (x8 en la parte de datos)
- es compatible con CAN 2.0

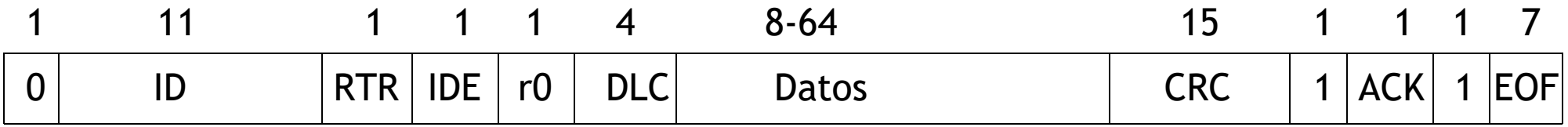
Arquitectura de CAN



La prioridad viene dada por el identificador

- menor número = mayor prioridad
- es habitual que cada nodo tenga su identificador
- pero también es posible que un nodo mande mensajes con distinto identificador
  - hay que asegurarse que estos son únicos

Estructura del frame (CAN 2.0A)



- ID: Identificador del mensaje
- RTR: Remote Transmission Request
- IDE: Extensión de identificador (0 si es de 11 bits)
- r0: reservado
- DLC: Longitud de datos en bytes
- CRC: Código de redundancia cíclico
- ACK: Confirmación (el transmisor manda un 1 pero el receptor puede cambiarlo a 0)
- EOF: End of Frame. 7 bits a 1 que marcan el final del frame y se usan para la sincronización de los nudos
- Bits de relleno (stuffing): para mantener la sincronización, cada 5 bits consecutivos iguales se añade uno distinto, descartado por el receptor
  - excepto en el EOF, ACK y el bit posterior al CRC
- Espaciado Interframe: 3 bits a 1 como mínimo

Arbitrio de prioridad

Es un arbitrio bit-a-bit, usando el identificador del nudo, que es único

El bus, eléctricamente, hace una operación *wire-and*

- el bit 0 es dominante
- el bit 1 es recesivo

Todos los nudos ponen en el bus a la vez sus identificadores y observan el resultado

- si es distinto de lo que escriben se retiran de la comunicación
- el que tenga el identificador más bajo "gana" y transmite el mensaje
- seis ciclos después de finalizarse el mensaje todos los que quieren transmitir lo vuelven a intentar

Ejemplo de arbitrio de prioridad

El nudo 15 transmite con identificador 0x0F

El nudo 16 transmite con identificador 0x1F

	start bit	ID Bits											Resto del frame
		10	9	8	7	6	5	4	3	2	1	0	
Nudo 15	0	0	0	0	0	0	0	0	1	1	1	1	...
Nudo 16	0	0	0	0	0	0	0	1	Cesa la transmisión				
CAN Bus	0	0	0	0	0	0	0	0	1	1	1	1	...

Gana el arbitrio el identificador más bajo (nudo 15)

- no hay colisiones

Protocolos CAN de alto nivel

Son necesarios para establecer la comunicación al nivel de aplicación

Los fabricantes de automóviles usan protocolos propios

Ámbito industrial

- CANopen
- CAN Kingdom
- DeviceNet
- SafetyBUSp

Hay muchos otros protocolos para otros ámbitos de aplicación

CAN RT Top: un desarrollo propio orientado a la comunicación entre tareas con prioridades seleccionables por el usuario

Ejemplo: Arquitectura para el sistema SCADA distribuido

Ejercicio 2.5. Plantear una arquitectura dirigida por eventos para el ejemplo SCADA distribuido

TTP

Es un protocolo de red basado en la arquitectura TTA (dirigida por el tiempo) y estandarizado como SAE Standard AS6003 2011

- en TTA todas las tareas son periódicas y las variables externas se muestrean periódicamente

Implementa un mecanismo de sincronización de relojes

- necesario para la noción de tiempo global de la arquitectura TT

Proporciona los servicios requeridos para implementar sistemas de tiempo real con tolerancia a fallos

- transmisión de mensajes predecible con confirmación
- detección de pertenencia
- cambios de modo rápidos
- redundancia

Estructura del sistema

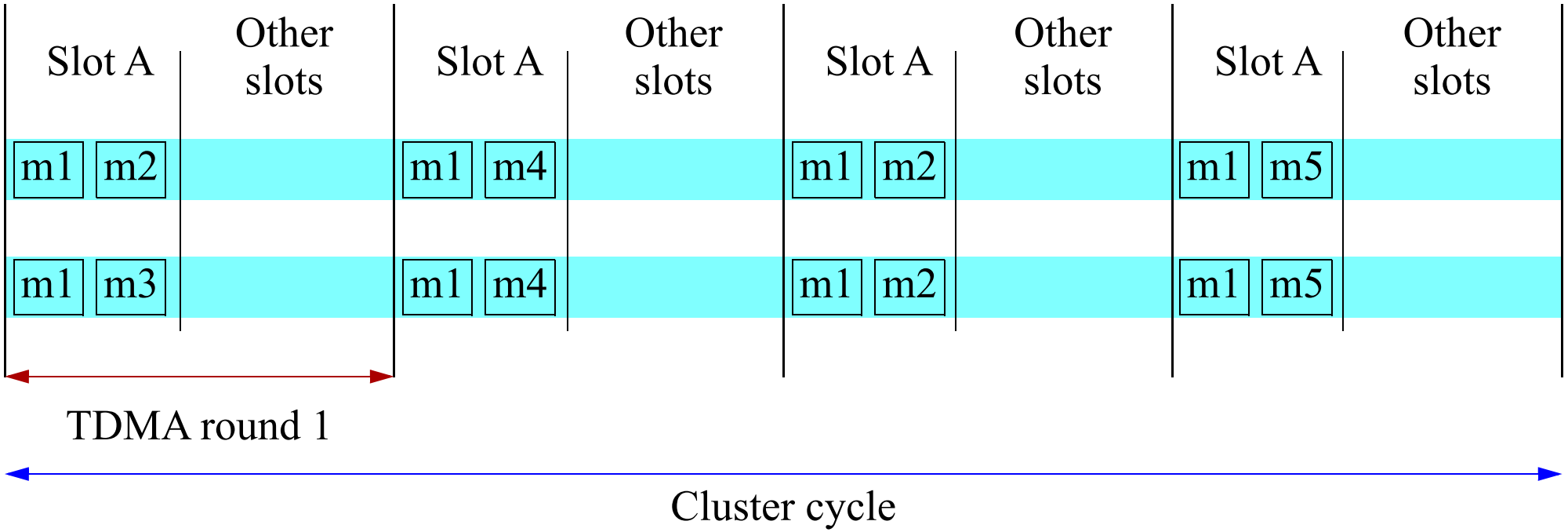
Nudos "fail-silent" con una red broadcast con dos canales redundantes

Los nudos se pueden replicar y agrupar en unidades tolerantes a fallos (FTU) que actúan como un único nudo virtual

Cada canal funciona por TDMA (Time Division Multiple Access) según un plan que se repite cíclicamente

- el ciclo es conocido como "cluster cycle"
- cada ciclo de cluster puede contener varios "TDMA round" en los que cada nudo tiene un "slot" para enviar un frame
  - dentro del ciclo cada TDMA round puede tener una estructura diferente
- en cada TDMA round cada nudo puede transmitir al menos una vez por canal
- cada frame puede contener uno o varios mensajes

Ejemplo de ciclo TTP



Sincronización de relojes

La precisión de la sincronización de relojes define el concepto de *macrotick*: granularidad temporal

- es configurable, del orden de pocos microsegundos (entre 1 y 10)

Todos los tiempos deben venir referidos a unidades medidas según esta granularidad

Al existir un concepto de reloj global

- el destino, origen o tipo de mensaje están predeterminados e implícitos por el instante de comienzo de la transmisión
- la detección de errores es fácil pues se sabe cuándo se deben recibir los mensajes

Características de TTP

El tamaño del frame varía entre 1 y 240 bytes de carga útil

La carga útil contiene mensajes que solo son conocidos y gestionados por la aplicación

- TTP solo gestiona frames

Los mensajes enviados por un canal y otro pueden ser distintos

- aunque se suelen replicar por motivos de tolerancia a fallos

Los frames son no expulsables

No hay fragmentación de mensajes grandes en paquetes más pequeños

Si el mensaje llega después del inicio de su ventana de transmisión no puede enviarse, incluso aunque haya tiempo disponible en su frame

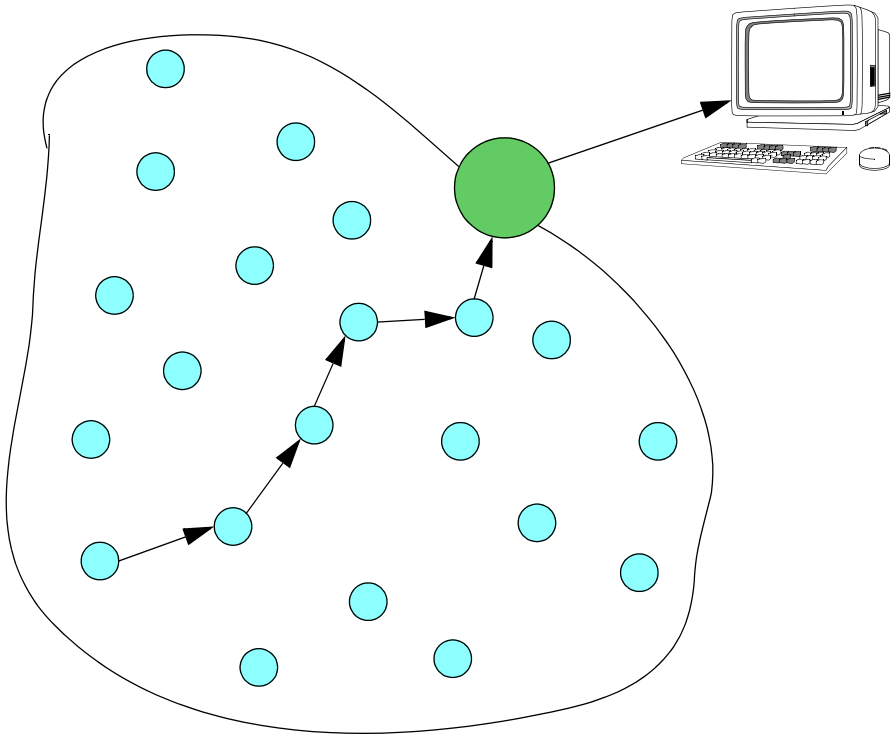
Ejemplo: Arquitectura para el sistema SCADA distribuido

Ejercicio 2.6. Plantear una arquitectura particionada para el ejemplo SCADA distribuido

Redes de sensores

Son conjuntos de sensores distribuidos espacialmente capaces de

- medir
- y actuar como puentes de la información



Hay redes capaces de enviar información bidireccionalmente

- capaces de actuar sobre los sensores (y/o actuadores)

Cada nudo es un sistema muy pequeño llamado "*mota*"

- potencia muy baja
- energía obtenida del ambiente o de una pila

Ámbitos de aplicación

Monitorización de recintos o instalaciones: seguridad

Monitorización de la salud o posición corporal

Monitorización del medio ambiente

- contaminación
- incendios forestales
- corrimientos de tierras
- calidad del agua
- inundaciones

Monitorización industrial

- estado de maquinaria
- condiciones ambiente de un proceso industrial
- salud estructural de grandes instalaciones (ej. presas)

Características de las redes de sensores

- Muy bajo consumo de energía
  - normalmente se conectan para una transmisión o recepción y se desconectan hasta el próximo momento de contacto
  - baterías y/o cosechado de energía (harvesting) solar, térmica, cinética
- Potencia de cálculo y memoria muy limitadas
- Habilidad de trabajar incluso con nodos que fallan
- Escalabilidad
- Tolerar condiciones ambientales extremas
- En algunos casos, heterogeneidad de nodos
- En algunos casos, movilidad de nodos
- Sistemas operativos muy pequeños
  - por ejemplo TinyOS, LiteOS o Contiki
- Arquitectura no centralizada y auto-reconfigurable

Protocolos de red para redes de sensores

En rápida evolución debido al fenómeno de IoT

- *IEEE 802.15.4* redes de ámbito personal (PAN, personal area networks)
  - capa física + MAC
  - problemas de escalabilidad y fiabilidad
  - tiempos de respuesta no acotados
- *IEEE 802.15.4e* red de malla inalámbrica (wireless mesh network)
  - topología de malla multisalto (multihop)
  - basada en la capa física de IEEE 802.15.4 (redefine la MAC)
  - mejora robustez ante interferencias externas
  - define modos con respuesta temporal acotable y requisitos de fiabilidad
- *ZigBee*, especificación propietaria de red de malla inalámbrica
  - basada en IEEE 802.15.4 (define servicios superiores)

Protocolos de red para redes de sensores (cont.)

- *WirelessHART*, especificación de red de malla inalámbrica
  - basada en IEEE 802.15.4 (define servicios superiores)
  - compatible con el bus de campo (cableado) HART
  - objetivos: fiabilidad y seguridad
- *ISA100.11a*
  - basada en IEEE 802.15.4, aunque define una MAC diferente y servicios superiores
  - objetivos: flexibilidad
- *Bluetooth smart*, también llamado low energy (LE)
  - disponible desde la especificación 4.0
- *Bluetooth mesh networking*
  - funciona sobre Bluetooth LE

Protocolos de red para redes de sensores (cont.)

- *Thread*, especificación de red de malla inalámbrica basada en IPv6
  - basada en 6LoWPAN, que a su vez se basa en IEEE 802.15.4
  - alta seguridad
  - promovida por Google
- *5G*
  - ¿el futuro?

Bibliografía

[1] "Software Engineering for Embedded Systems: Methods, Practical Techniques, and Applications". Robert Oshana, Mark Kraeling. Newnes, 2013

[2] "Embedded System Design". Springer, 2010

[3] "Avionics Application Software Standard Interface. ARINC Specification 653-1". Airlines Electronic Engineering Committee, Aeronautical Radio INC. March (2006).

[4] "AUTOSAR – A Worldwide Standard is on the Road". Simon Fürst, BMW Group. http://www.win.tue.nl/~mvdbrand/courses/sse/0910/AUTOSAR.pdf

[5] AUTOSAR: http://www.autosar.org

[6] "Intrinsic Safety of AUTOSAR Basic Software". Vector. http://vector.com/portal/medien/cmc/press/Vector/AUTOSAR_TTTech_ElektronikAutomotive_201204_PressArticle_EN.pdf

[7] "Real-Time Systems: Design Principles for Distributed Embedded Applications". Hermann Kopetz. Springer, 2011

[8] "TTP Communication Protocol". SAE. SAE Standard AS6003. 2011.

Bibliografía (cont.)

[9] Wikipedia. https://en.wikipedia.org/