Actualización de Servicios

Unidad 2
Actualización de *software*

Índice

Bibliografía

[DD68] Robert C. Daley, Jack B. Dennis: “Virtual Memory, Processes, and Sharing in MULTICS”. Commun. ACM 11(5): 306-312 (1968)

[GIL78] Hannes Goullon, Rainer Isle, Klaus-Peter Löhr: “Dynamic Restructuring in an Experimental Operating System”. IEEE Trans. Software Eng. 4(4): 298-307 (1978)

[HN05] Michael W. Hicks, Scott Nettles: “Dynamic software updating”. ACM Trans. Program. Lang. Syst. 27(6): 1049-1096 (2005)

[MBM12] Emili Miedes, Josep M. Bernabeu-Aubán, Francesc D. Muñoz-Escoí: “Software Adaptation through Dynamic Updating”, informe técnico ITI-SIDI-2012/010, Universitat Politècnica de València, septiembre 2012.

[SAM13] Habib Seifzadeh, Hassan Abolhassani, Mohsen Sadighi Moshkenani: “A survey of dynamic software updating”. Journal of Software: Evolution and Process 25(5): 535-568 (2013)

[SF93] Mark E. Segal, Ophir Frieder: “On-the-Fly Program Modification: Systems for a Dynamic Updating”. IEEE Software 10(2): 53-65 (1993)

Objetivos

- Distinguir entre técnicas de actualización de programas estáticas y dinámicas.
- Conocer y aplicar las técnicas de actualización dinámica más importantes.
- Identificar la conversión de estado como el problema más delicado en una actualización dinámica.
- Revisar algunos sistemas relevantes en la evolución de los sistemas de actualización dinámica.

Índice

1. Introducción

- Las aplicaciones informáticas necesitan actualizarse. Motivos:
  - Extensión de su funcionalidad.
  - Eliminación de errores.
  - Mejora de su eficiencia.
  - Eliminación de vulnerabilidades.

1. Introducción

- Tradicionalmente, la actualización ha sido estática ("*stop-and-restart*"):
  - Los procesos que ejecuten la aplicación son parados.
  - Se elimina la versión anterior.
  - Se instala la nueva versión.
    - Así como todos los demás módulos (bibliotecas, servicios complementarios) de los que dependa.
  - Se inicia la ejecución de la nueva versión.

1. Introducción

- Una actualización estática:
  - Impide que el servicio siga estando disponible.
    - Durante el intervalo invertido en la actualización.
  - Requiere la intervención de algún administrador del sistema en el que se ejecute la aplicación.

1. Introducción

- Deberían utilizarse mecanismos de actualización dinámica:
  - Que minimicen los intervalos de indisponibilidad.
  - Que puedan automatizarse.

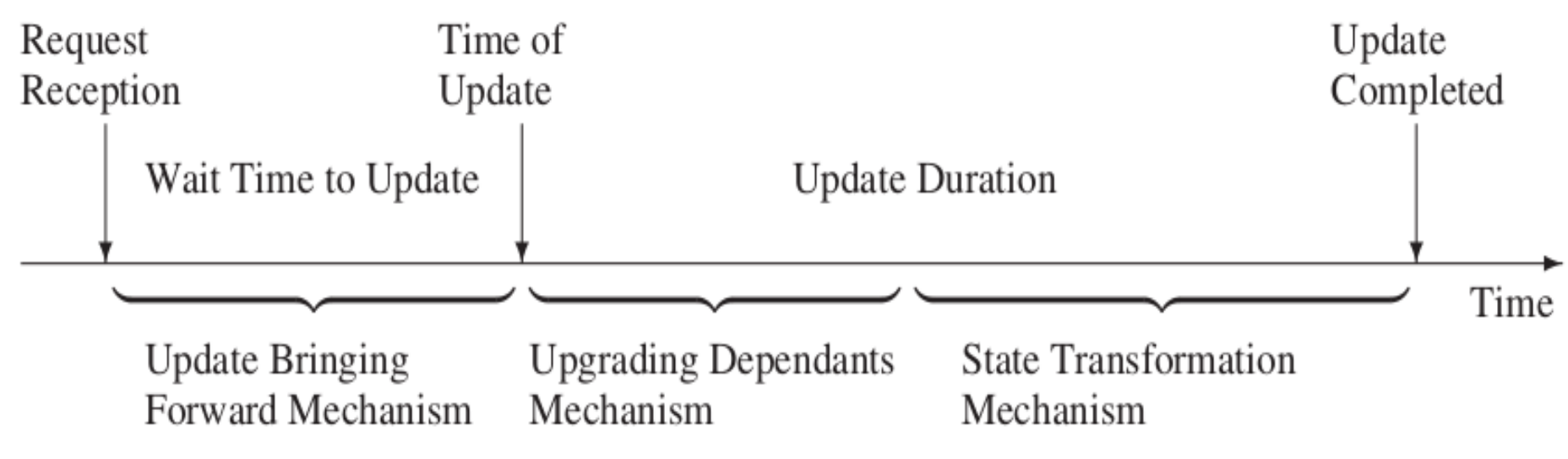
Índice

2. Actualización dinámica

- Según [HN05], la actualización dinámica debe ofrecer estas características:
  - Flexibilidad. Cualquier parte del sistema puede ser actualizada sin incurrir en indisponibilidad.
  - Robustez. El sistema debe minimizar el riesgo de errores y caídas durante la actualización, utilizando mecanismos automatizados que aseguren su corrección.
  - Facilidad de uso. De nuevo, para evitar errores durante la actualización.
  - Baja sobrecarga. No debería haber impacto sobre el rendimiento del sistema.

2. Actualización dinámica

- En [SAM13] se distinguen estas fases:



2. Actualización dinámica

- Eventos importantes:
  1. Petición
     - El sistema recibe la petición de actualización.
     - No siempre es posible empezar inmediatamente:
       - Esperar que concluyan las invocaciones en curso.
         - O que lleguen a un punto donde no haya riesgo de inconsistencias.
       - ¿Bloquear inicio de nuevas invocaciones?
     - Inicia la fase: “Mecanismo de preparación de la actualización”.

2. Actualización dinámica

2.Instante de actualización

- Finaliza la preparación y empieza la fase de “actualización de dependientes”.
  - Dependiente: Aquel subprograma que necesite invocar al programa, módulo o rutina actualizados.
- Esta segunda fase podrá solaparse con la fase de “transformación de estado”.
- El mecanismo de “transformación de estado” también incluye una transferencia de estado.
  - La nueva versión suele ubicarse en una región de memoria diferente.

2. Actualización dinámica

3.Fin de actualización

- Cuando haya finalizado todo el proceso:
  - Gestión de dependencias.
  - Transferencia de estado.
  - Transformación de estado.
- La nueva versión seguro que será capaz de recibir invocaciones desde sus “dependientes”.
- Para que un sistema de actualización sea realmente dinámico...
  - ...entre los eventos 1 y 3 debe permitirse que lleguen y se atiendan nuevas peticiones.
  - ¡No siempre será posible!!!

Índice

3.Métricas

- En [SAM13] se identifican 8 métricas para caracterizar las técnicas de actualización dinámica:
  - Unidad de actualización.
  - Actualización de módulos dependientes.
  - Atomicidad.
  - Transformación de estado.
  - "*Type-safety*".
  - Instante de actualización.
  - Actualización de código activo.
  - Diferenciación de código.

3.1. Unidad de actualización

- Estudia qué componentes de la aplicación o servicio necesitan ser cargados en memoria principal para completar la actualización.
- Tres alternativas:
  - Recarga completa. Todo el programa necesita ser reemplazado.
  - Recarga de módulos. Sólo los módulos modificados necesitan recargarse.
  - Recarga de unidades. La aplicación se divide en unidades que a su vez pueden estar compuestas por módulos.
    - Se recarga toda unidad con algún módulo modificado.

3.2. Actualización de módulos dependientes

- Esta métrica se refiere al mecanismo que utilizarán los módulos que puedan invocar a un módulo actualizado para adaptarse a su nueva ubicación.
  - Tiene sentido en caso de que en la métrica anterior se utilice:
    - Recarga de módulos.

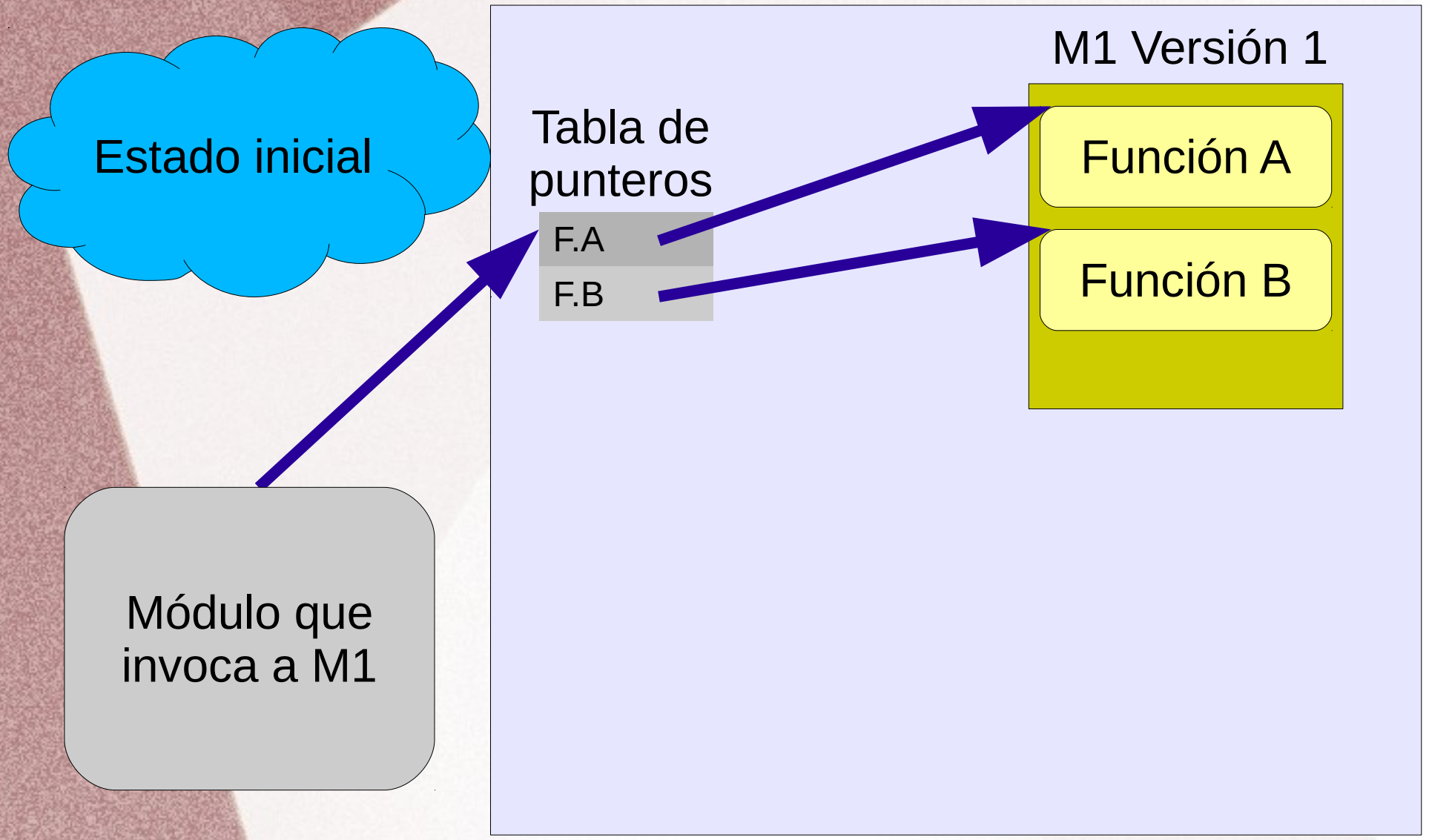
3.2. Actualización de módulos dependientes

- Se discuten soluciones de bajo nivel.
- Cuatro alternativas:
  - Uso de un nivel de indirección.
  - Redirección.
  - Reemplazo en la misma ubicación.
  - Actualización de los invocadores.

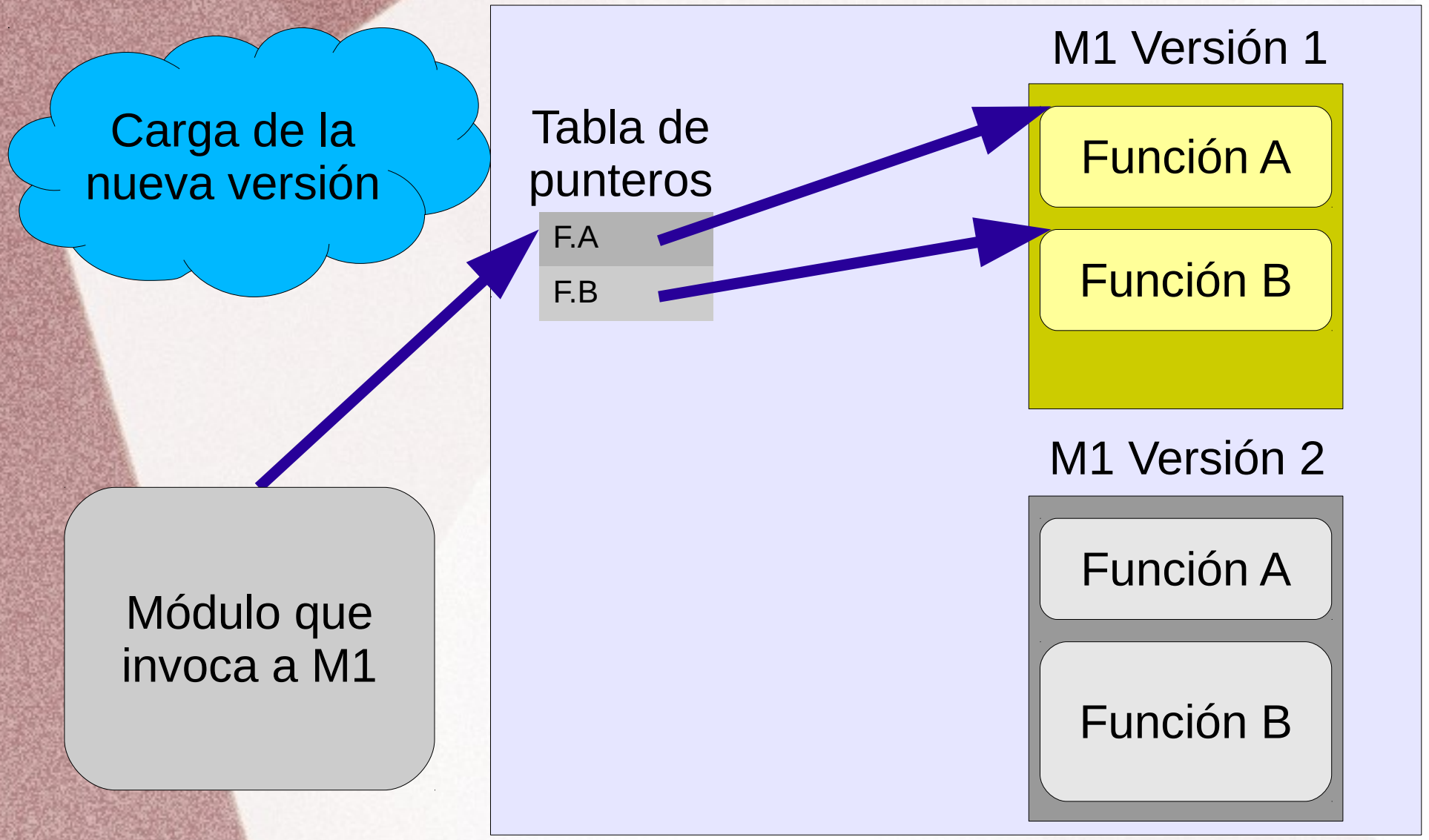
3.2.1. Indirección

- Es el mecanismo habitualmente utilizado en las bibliotecas de enlace dinámico.
- Existe una tabla con las direcciones de las rutinas del módulo a actualizar.
- Cuando otros módulos necesiten invocar a una rutina, llaman a la dirección contenida en una de las posiciones de la tabla.
- Si el módulo es actualizado...:
  - Se podrá cargar en otra región disponible.
  - Se actualizarán todas las entradas de la tabla, para recoger sus nuevas ubicaciones.
  - Si no se necesitase transformar el estado, podrían convivir ambas versiones mientras las llamadas ya iniciadas sobre la versión anterior no concluyan.

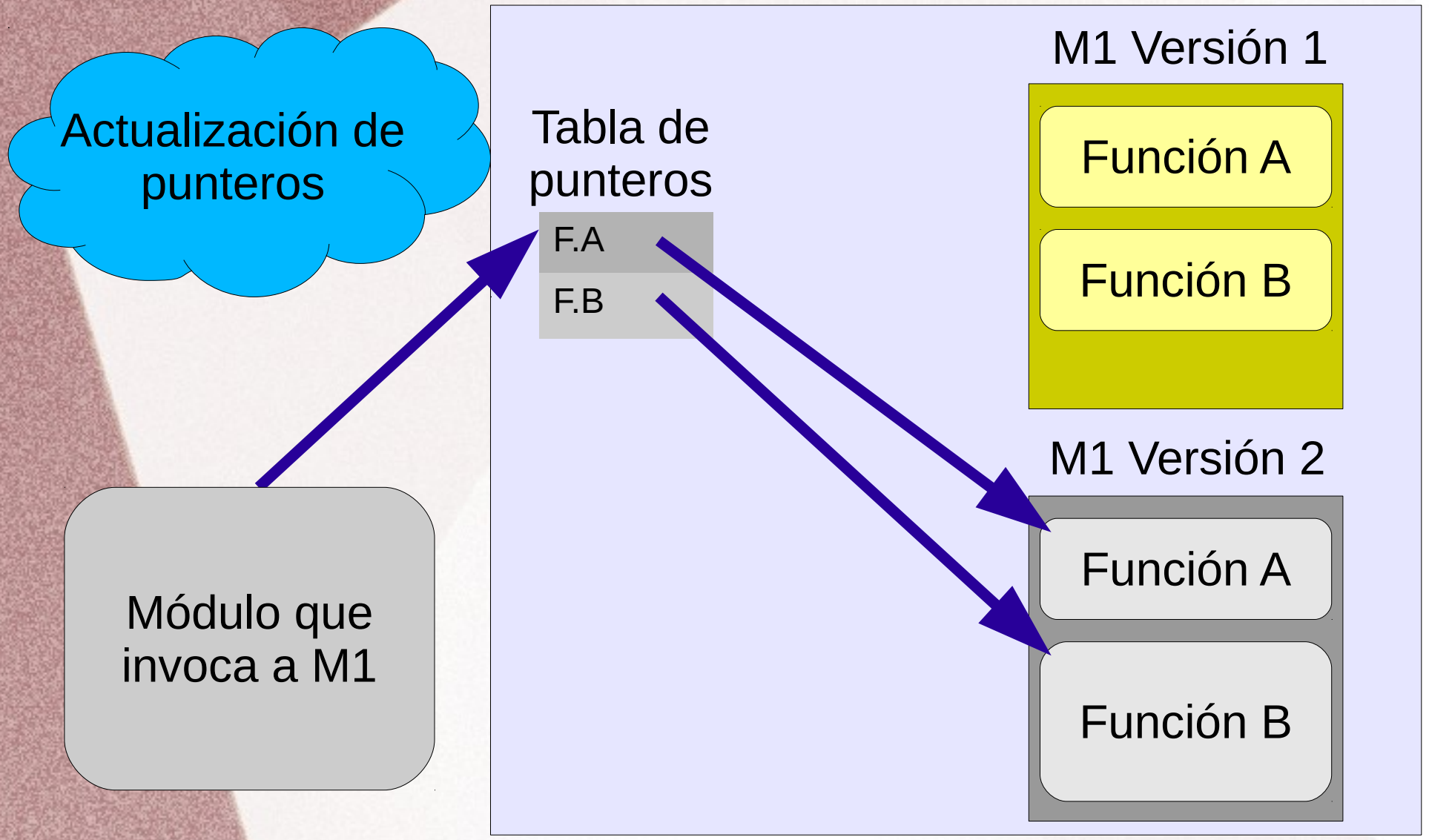
3.2.1. Indirección



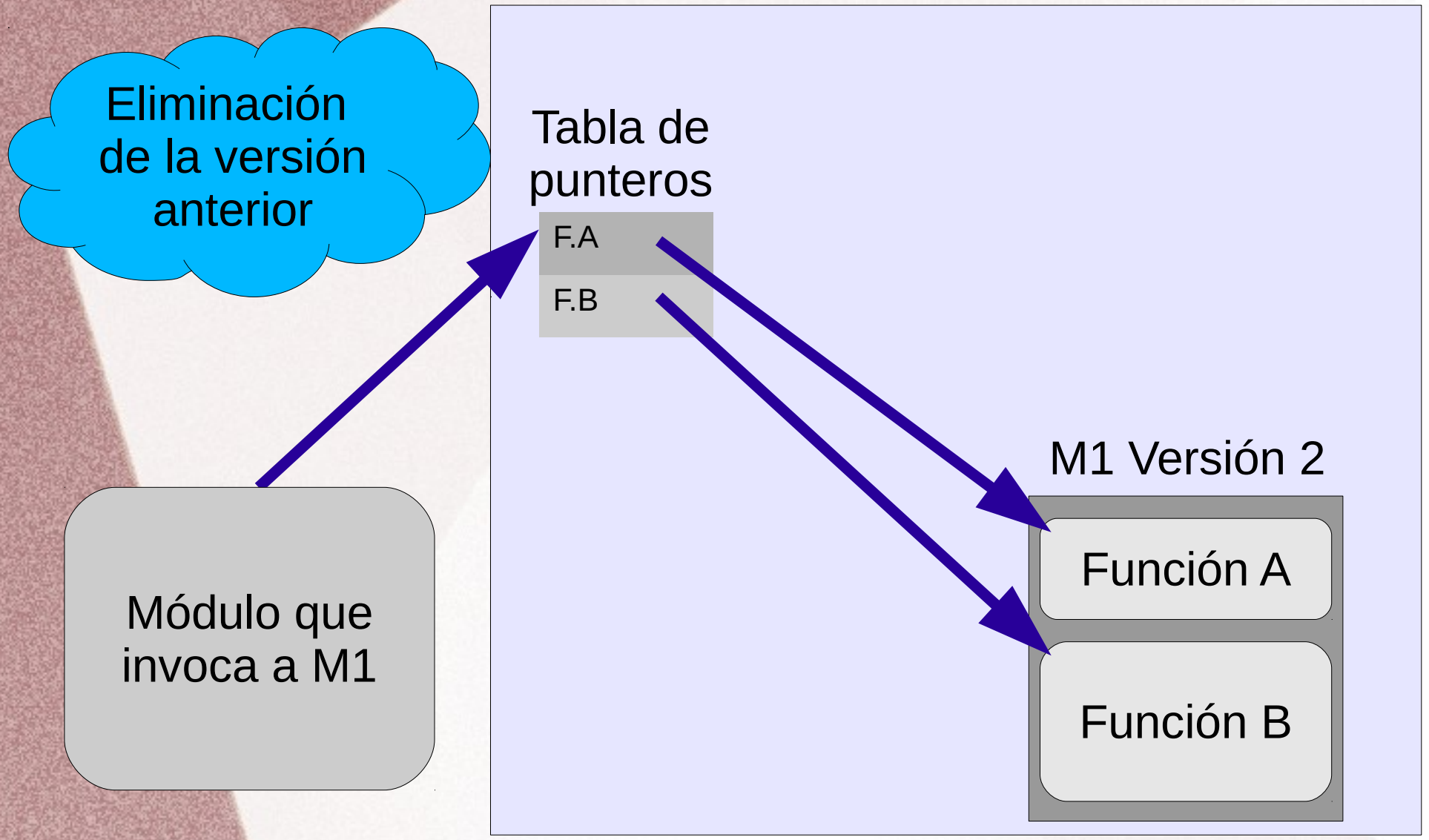
3.2.1. Indirección



3.2.1. Indirección



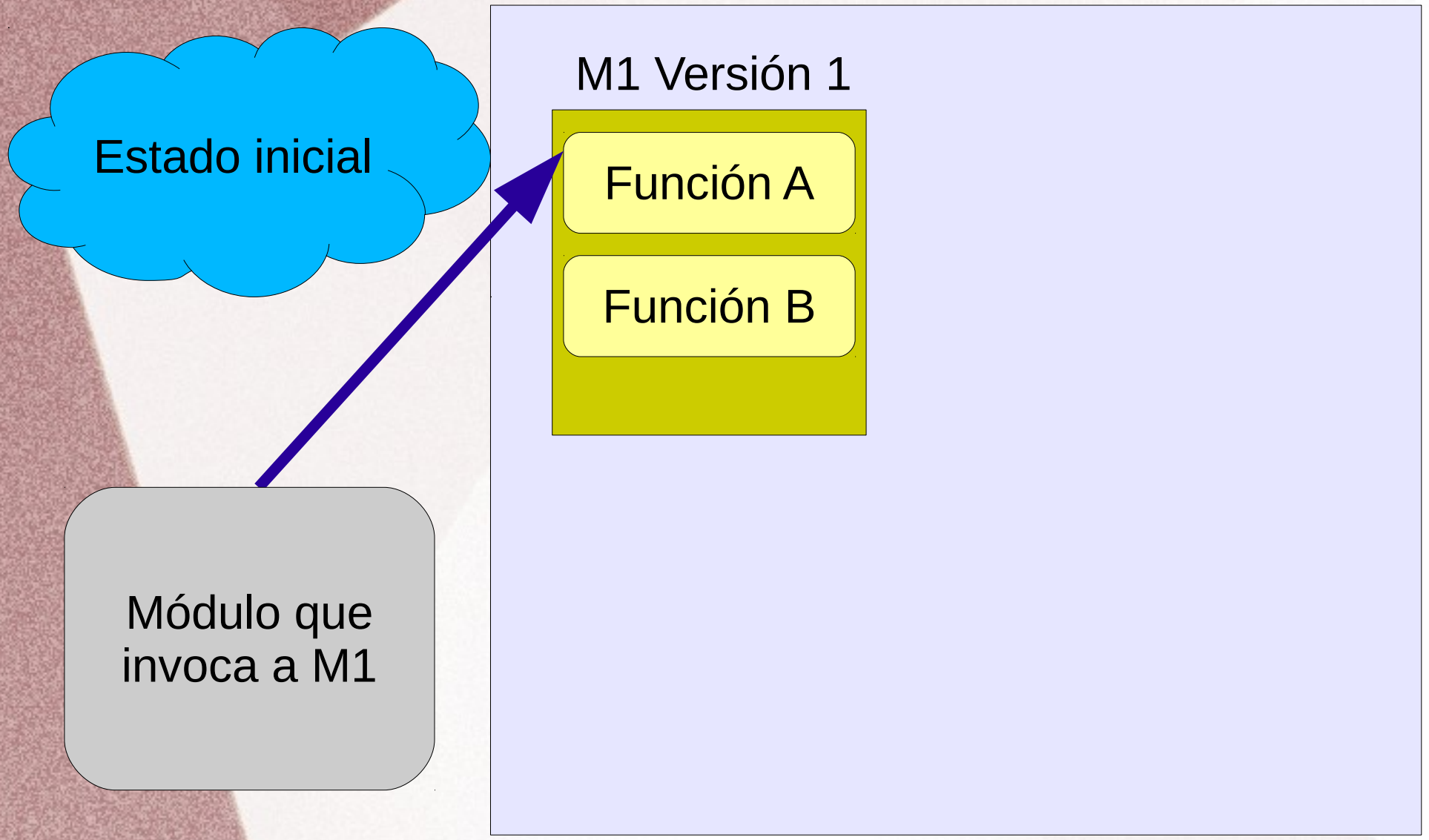
3.2.1. Indirección



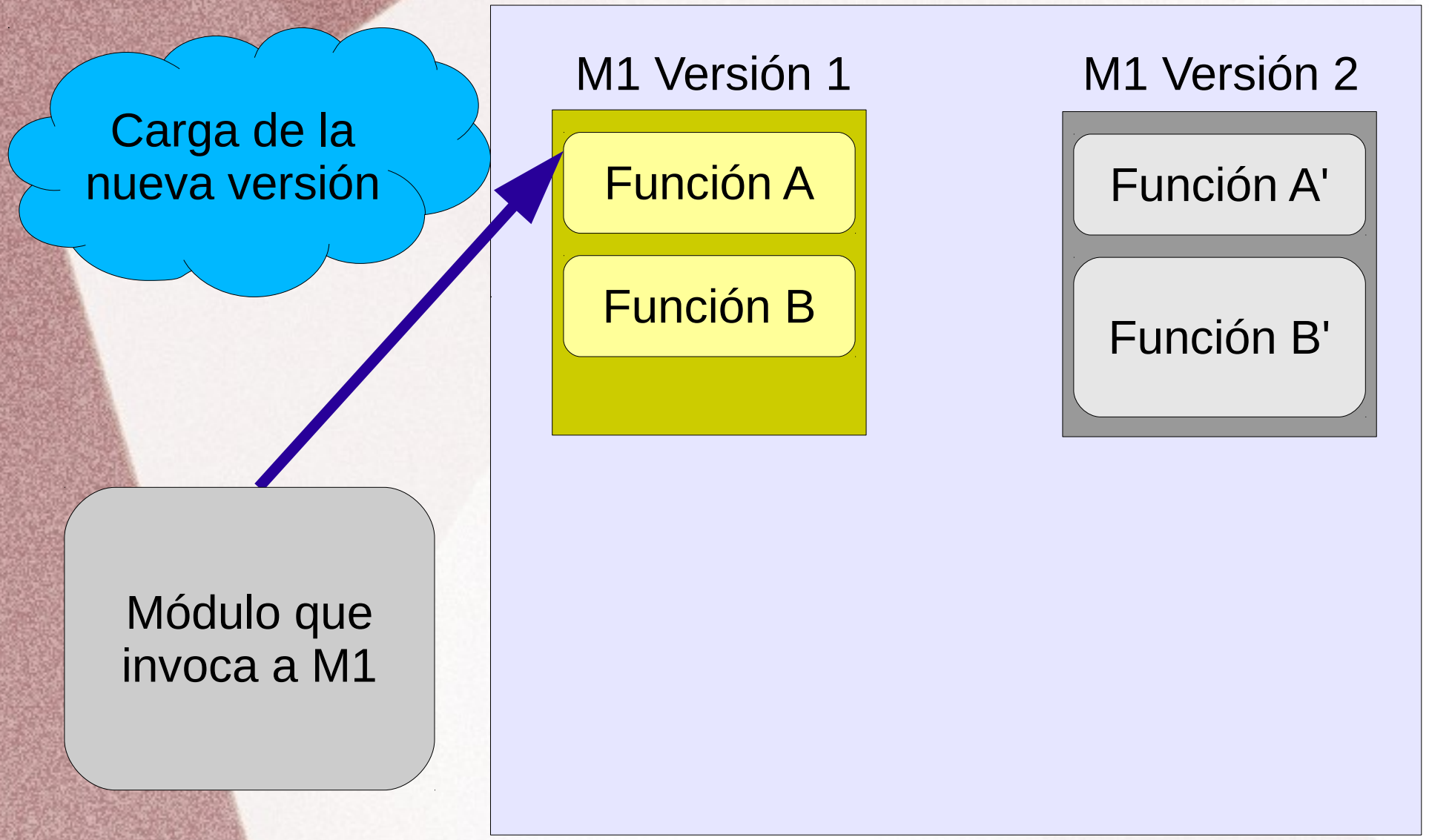
3.2.2. Redirección

- Se sustituye la primera o primeras instrucciones de cada función del módulo original por instrucciones de salto a sus nuevas posiciones en el nueva versión del módulo.
  - No permite una eliminación total de la versión antigua.
  - Exige mantener la misma interfaz.

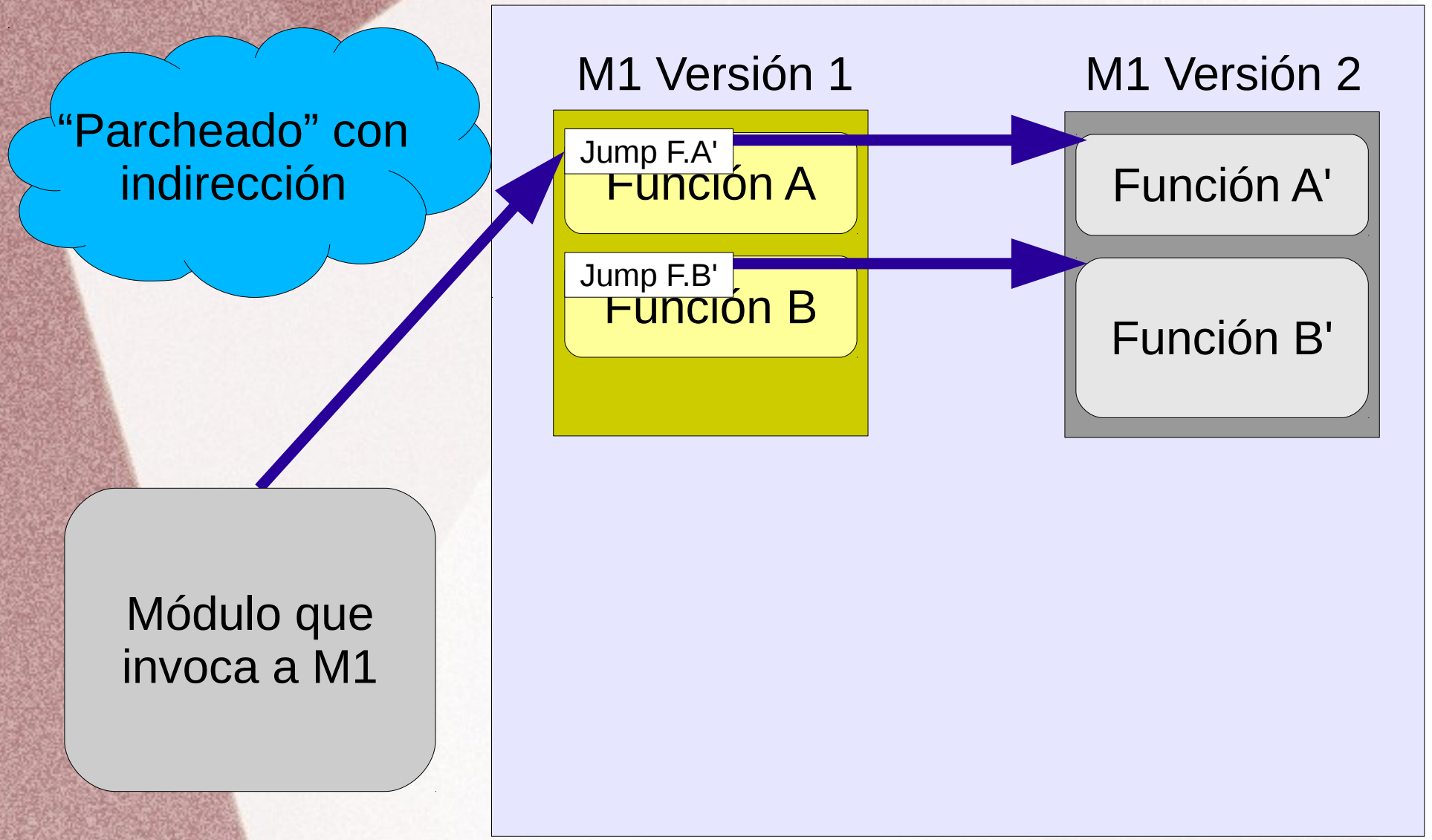
3.2.2. Redirección



3.2.2. Redirección



3.2.2. Redirección



3.2.3. Reemplazo en misma ubicación

- El módulo nuevo pasa a ocupar las mismas direcciones (virtuales) que su versión anterior.
  - Necesita que no haya ningún hilo de ejecución utilizando la versión antigua.
  - Requiere que el nuevo módulo quepa en el hueco inicialmente destinado a la versión anterior.
- Al igual que en las dos alternativas anteriores, no se exige realizar ningún cambio en los módulos invocadores.

3.2.4. Actualización de invocadores

- Se revisa el código presente en memoria.
  - Asumiendo que el módulo actualizado pertenece a una sola aplicación.
- Se cambia la dirección antigua por la nueva en todas las instrucciones de llamada a subprograma donde apareciese la antigua.
  - Requiere revisar todo el código de la aplicación.
  - No admite ningún cambio en la interfaz del subprograma actualizado.

3.3. Atomicidad

- La “atomicidad” implica que mientras se realice la actualización se estará prohibiendo la actividad de la aplicación.
  - Implica “exclusividad”: todos los esfuerzos se dedican a la actualización.
  - Implica también indisponibilidad mientras dure la actualización.

3.3. Atomicidad

- Tres niveles:
  - Atómico: La actualización se realiza en una única etapa.
    - Sin disponibilidad de servicio.
  - No atómico: La actualización se realiza en paralelo con la actividad normal de la aplicación.
    - Puede haber inconsistencias si se altera la estructura de algún tipo de datos.
  - Parcialmente atómico (nivel recomendado):
    - El código que no entrañe riesgo de inconsistencias se actualiza en paralelo con la ejecución normal.
    - El resto se ejecuta en múltiples subetapas atómicas.

3.4. Transformación de estado

- Esta métrica analiza el conjunto de mecanismos necesarios para que el estado de la versión anterior del programa pueda ser utilizado por su nueva versión.
  - Aunque no se hayan modificado las estructuras de datos, como mínimo habrá que moverlo a una nueva ubicación.
    - La nueva versión suele ocupar una región distinta del espacio de memoria.

3.4. Transformación de estado

- Se distinguen dos tareas:
  - Transferencia. Conseguir que la nueva versión reciba el estado de la anterior.
  - Transformación. Adaptar ese estado para que pueda ser utilizado por la nueva versión.
    - Cambios en los tipos de datos.
    - Cambios en los valores.

3.4. Transformación de estado

- Se distinguen seis ejes en esta métrica:
  1. Cómo transferir el estado.
  2. Cómo transformar.
  3. Qué transferir.
  4. Cuándo transferir.
  5. Qué transformar.
  6. Dónde transformar.

3.4.1. Cómo transferir

- Describe cómo realizar la transferencia de estado entre versiones.
- Dos alternativas:
  - Copia: Los valores de las estructuras de datos son copiados a su nueva ubicación.
  - Mantenimiento ("*shadow*"):
    - El estado se mantiene en su ubicación original, conocida por la nueva versión.
    - Pueden necesitarse punteros a nuevas estructuras adicionales.

3.4.2. Cómo transformar

- El estado necesita ser transformado o adaptado si la nueva versión utiliza distintos tipos o estructuras de datos.
- Dos alternativas:
  - Transformadores especificados por el programador.
    - Son ejecutados durante la actualización.
  - Reejecución. Se hace un "*rollback*" hacia un punto previo de la ejecución en que el estado sea coherente entre ambas versiones.
    - A partir de ese punto se ejecuta la nueva versión, empezando por la secuencia sobre la que se aplicó el "*rollback*".

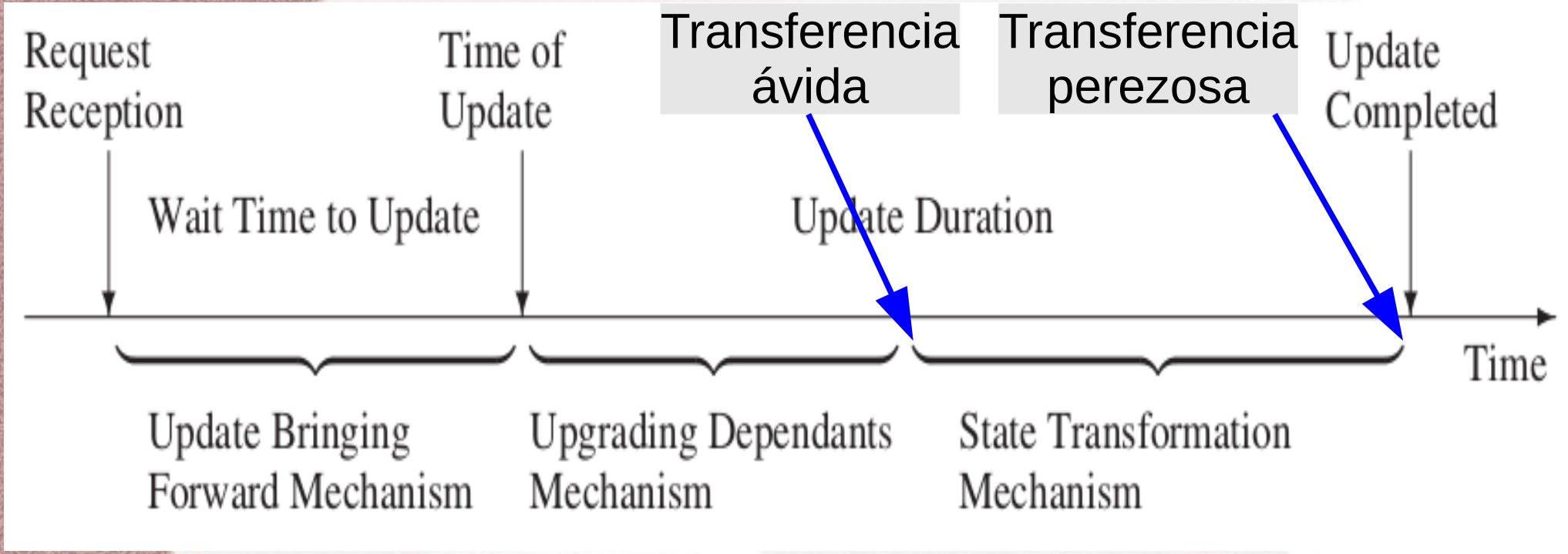
3.4.3. Qué transferir

- Especifica qué estado debe transferirse.
- Dos categorías:
  - Estado global: Las variables u objetos de ámbito global.
    - Será el estado completo si ha habido una etapa inicial de bloqueo de nuevas invocaciones y espera de finalización de las invocaciones en curso.
  - Estado activo: Las variables u objetos de ámbito local, asociados a las invocaciones activas.

3.4.4. Cuándo transferir

- Especifica el instante en que el estado es transferido.
- Dos alternativas:
  - Ávida (“*eager*”). La transferencia se realiza tan pronto como se hayan adaptado los módulos dependientes.
  - Perezosa (“*lazy*”). La transferencia se retrasa hasta el momento en que la primera invocación a la nueva versión es recibida.
    - La transformación habrá concluido previamente.

3.4.4. Cuándo transferir



- La transferencia perezosa puede reducir el intervalo de actualización.
  - No hay frontera (sincronización) entre las dos etapas anteriores, que podrán solaparse.

3.4.5. Qué transformar

- Esta quinta métrica distingue entre:
  - Valores: Existen ciertas reglas para modificar el valor de algunas variables o atributos.
    - Pero sus tipos / clases se mantienen.
  - Tipos: No sólo se modifican algunos valores sino también los tipos / clases de algunas variables u objetos.
    - Ejemplo: Añadir o eliminar atributos a alguna clase de objeto o estructura.

3.4.6. Dónde transformar

- Especifica si la transformación requerirá o no un almacén o formato intermedio.
- Dos alternativas:
  - Directa: La transformación no requiere ningún buffer intermedio.
    - Se transfiere directamente de origen (versión antigua) a destino (nueva).
  - Indirecta: La transformación requiere un formato estándar intermedio y cierto almacén o buffer para guardarlo.

3.5.”Type-safety”

- Un programa cumple con esta métrica si los tipos que utiliza o retorna son los esperados por sus invocadores.
- La métrica puede incumplirse si:
  - Se modifican las interfaces de un módulo actualizado.
    - Y no se actualizan sus invocadores.
  - La atomicidad (1ª alternativa en la 1ª métrica) lo resuelve si los cambios fueran internos a un programa.

3.5. “Type-safety”

- Cuatro grados:
  - No hacer nada.
  - Uso de “stubs”.
  - Reordenación de actualizaciones.
  - Conjuntos consistentes.

3.5.1. No hacer nada

- Se asume que el programador organizará por sí mismo la actualización.
  - El sistema de actualización no tendrá que preocuparse.
    - De ahí el nombre de esta variante.
  - Cuando haya un cambio en una interfaz...:
    - El programador modificará también todos los componentes que la puedan invocar.
    - Ordenará adecuadamente los pasos de actualización para que no se genere ninguna incoherencia.
      - Gestionando los bloqueos pertinentes para que no haya inconsistencias.
      - Habra indisponibilidad temporal.

3.5.2. Uso de “stubs”

- Ocultar los cambios en las interfaces mediante “stubs”.
  - El “stub” ofrecerá a los invocadores la interfaz de la versión anterior.
  - Realizará las transformaciones adecuadas para invocar la nueva interfaz.
  - Adaptará también los resultados o los argumentos de salida.

3.5.3. Reordenación

- El subsistema de actualización no inicia de inmediato la gestión de todas las peticiones de actualización.
  - Cuando haya cambios en interfaces.
- Examina las dependencias entre todos los módulos a actualizar y decide qué secuencia es la más apropiada.
  - Al igual que en la primera solución, normalmente necesitará introducir bloqueos.
    - Indisponibilidad → ¡No dinámico!!

3.5.4. Conjuntos consistentes

- Objetivo similar al anterior:
  - Los módulos a actualizar se estructuran en conjuntos.
  - Cada conjunto agrupa varios módulos con interdependencias.
  - Las interfaces modificadas no resultan visibles a módulos no incluidos en el conjunto.
  - Cada conjunto se actualiza atómicamente.

3.6. Instante de actualización

- Debe encontrarse un equilibrio entre inmediatez y consistencia.
- Variantes:
  - Actualización inmediata.
    - Prioriza el instante frente a la consistencia.
  - Actualización diferida.
    - Asigna mayor prioridad a la consistencia.

3.6.1. Actualización inmediata

- Tres variantes:
  - Estricta: Sólo se aplica cuando el módulo a actualizar esté inactivo.
    - Si hay actividad, se aborta la actualización.
  - Multiversionada: Las invocaciones activas siguen su curso. Las nuevas utilizan la nueva versión.
    - Concurrencia. Gestión difícil de la consistencia.
  - Sobre código activo: Ver métrica 3.7.
    - Múltiples dificultades.

3.6.2. Actualización diferida

- Dos dimensiones:
  - Cómo determinar el instante apropiado:
    1. Finalización de actividad.
    2. Especificación de puntos seguros.
    3. Inactividad (“quiescence”).
    4. Comprobación periódica.
  - Cómo llegar a ese instante:
    5. Rechazo de peticiones.
    6. Relajación.

3.6.2.1. Finalización

- La actualización se iniciará cuando todas las invocaciones hayan finalizado.
  - Esto puede comprobarse cuando cada invocación retorne al invocador.
  - Se prohíbe la actualización mientras haya tareas activas en el módulo o módulos a actualizar.

3.6.2.2. Puntos seguros

- El programador define puntos del código en los que un inicio de la actualización no supondría ningún riesgo para la consistencia.
- Cuando todas las invocaciones activas estén en esos puntos seguros, la actualización empieza.
  - Las invocaciones permanecen paradas en esos puntos.

3.6.2.3. Inactividad

- Similar a la primera alternativa, pero ligeramente más relajada.
  - Se exige que los módulos a actualizar estén inactivos.
  - No exige que todas las invocaciones hayan terminado.
    - Las que haya pueden estar paradas en un punto seguro.

3.6.2.4. Comprobación periódica

- El sistema de actualización comprueba periódicamente si ha habido alguna solicitud de actualización y el nivel de actividad en los módulos permite llevarla a cabo.

3.6.2.5. Rechazo de peticiones

- Primera variante dentro de la segunda dimensión.
- Objetivo: Llegar cuanto antes al instante en que la actualización se permitirá.
- Mecanismo: Rechazar todas las nuevas invocaciones desde el momento en que se haya solicitado una actualización.
  - Así, las activas conseguirán terminar antes.
  - Menor concurrencia en el acceso a recursos compartidos.
    - Menor probabilidad de bloqueo.
- Problema: Introduce indisponibilidad.

3.6.2.6. Relajación

- Se fuerza a que el programador introduzca puntos seguros de actualización adicionales.
  - Así será más fácil que cada invocación activa llegue a alguno de ellos.
  - Se sigue admitiendo la entrada de nuevas invocaciones.
    - No perjudica tanto a la disponibilidad.

3.7. Actualización de código activo

- Implica que la actualización se pueda llevar a cabo en cualquier momento.
  - Aunque haya invocaciones en curso.
  - Sin necesidad de rechazarlas (en su inicio), pararlas o abortarlas.
  - Única forma de asegurar una disponibilidad máxima.
    - Objetivo: flexibilidad.

3.7. Actualización de código activo

- Dos dificultades / dimensiones:
  - En qué punto de la ejecución realizar la actualización.
    - Migrando las actividades.
  - Cómo transferir y transformar el estado a la nueva versión.
    - Mecanismos enunciados en la sección 3.4.

3.7. Actualización de código activo

- Soluciones para la 1ª dimensión:
  - Puntos seguros: Acompañados por tablas de correspondencia.
    - Indican a qué (nuevo) subprograma invocar desde ese punto.
  - Bloqueo: Uso de puntos seguros, que impliquen bloqueo de actividad, permitiendo su actualización.
  - Actualización inmediata ("*just-in-time*"): Necesita mismo estado en las dos versiones.

3.8. Diferenciación de código

- En esta métrica se evalúa qué mecanismo utiliza el sistema de actualización para averiguar (mediante comparación de versiones) qué partes del programa hay que actualizar.
- Cuatro alternativas:
  - A partir de código objeto.
  - A partir de código fuente.
  - A partir de estructuras de alto nivel.
  - Especificación explícita por parte del programador.

Índice

4. Ejemplos

- Todavía no hay sistemas de actualización completamente dinámicos:
  - Sigue habiendo intervalos de indisponibilidad.
    - Cada vez más breves.
  - El sistema no es completamente transparente.
    - El programador debe ser consciente de la existencia del sistema de actualización.
  - La actualización no siempre está automatizada.

4. Ejemplos

- Realizaremos un breve recorrido histórico (consultar [SF93]):
  - MULTICS (1965).
  - DAS (1978).
  - Servicios distribuidos.

4.1. MULTICS

- MULTICS [DD68] fue uno de los primeros sistemas con soporte para enlace dinámico.
- El enlace dinámico no garantiza actualización dinámica, pero...:
  - Proporciona indirección (3.2.1).
    - Permite actualizar dependencias.
  - Permite recarga de módulos (3.1).
  - Ofrece buena base para actualización parcialmente atómica (3.3).
    - Pero se implantó atómica. Sin transformación de estado.
- ¡Diseñado en 1965!! ¡Actualización estática!!

4.2. DAS

- DAS [GIL78] fue un sistema operativo experimental que...:
  - Extendía los principios del enlace dinámico para sustituir los módulos enlazados en tiempo de ejecución.
    - Mecanismos dependientes de la MMU de los PDP-11.
    - Migración difícil a otras arquitecturas.
  - Admitía cambios menores en interfaces.
    - Conversiones de tipos.

4.2. DAS

- Cómo gestiona cada métrica:
  1. Unidad: Módulo.
  2. Dependencias: Indirección.
  3. Atomicidad: Parcialmente atómico.
  4. Transformación: Sí. Solución en cada dimensión:
     - Copia. Programador. Global. Perezosa. Tipos. Directa/Indirecta (ambas, según tipo).
  5. “Type-checking”: No se admiten cambios mayores en la signatura / interfaz del módulo.
  6. Instante: Actualización diferida. Finalización. Admite peticiones en todo momento (gestionadas por contador).
     - Comenta opción de multiversionado, pero no lo describe.
  7. Código activo: No aplicable (ver criterio 6).
  8. Diferenciación: No documentado.

4.3. Servicios distribuidos

- Objetivo de esta asignatura.
- Se analizarán en detalle en la unidad 3.
- Existen propuestas desde la década de los 80 del siglo pasado: Argus, publicaciones de Segal y Frieder...

Índice

5. Resultados de aprendizaje

- Al finalizar esta unidad, el alumno debe ser capaz de:
  - Conocer los objetivos de todo sistema de actualización dinámica: robustez, disponibilidad, facilidad de uso y mínima sobrecarga.
  - Conocer y aplicar los mecanismos de actualización dinámica básicos.
  - Caracterizar cualquier sistema de actualización actual (con las métricas).
  - Identificar sus principales aportaciones e inconvenientes.