Actualización de Servicios

Unidad 3
Actualización de servicios replicados

Índice

1. Introducción
2. Replicación
3. Revisión histórica
4. Propuestas
5. Resultados de aprendizaje

Bibliografía

[ALS06] Sameer Ajmani, Barbara Liskov, Liuba Shrira: “Modular Software Upgrades for Distributed Systems”. ECOOP 2006: 452-476

[BD92] Toby Bloom, Mark Day: “Reconfiguration in Argus”. 1st Intnl. Wshop. On Configurable Distrib. Syst. (CDS) 1992: 176-187

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

[SM02] Marcin Solarski, Hein Meling: “Towards Upgrading Actively Replicated Servers On-the-Fly”. COMPSAC 2002: 1038-1046

[TMM01] L. A. Tewksbury, Louise E. Moser, P. M. Melliar-Smith: “Live Upgrades of CORBA Applications Using Object Replication”. ICSM 2001: 488-

Objetivos

- Conocer y utilizar las ventajas que aporta la replicación de componentes para realizar una actualización dinámica.
- Aprender y aplicar algunos esquemas de actualización dinámica para servicios replicados.
- Revisar algunos sistemas relevantes en la evolución de los sistemas de actualización dinámica sobre servicios replicados.

Índice

1.Introducción

2.Replicación

3.Revisión histórica

4.Propuestas

5.Resultados de aprendizaje

1. Introducción

- La unidad 2:
  - Caracterizó los sistemas de actualización dinámica.
  - Presentó las métricas a considerar para evaluar un sistema de este tipo.
    - Esas métricas se especificaron para sistemas formados por un solo ordenador.
    - ¿Qué ocurre en un sistema distribuido?

1. Introducción

- Un sistema distribuido estará compuesto por múltiples ordenadores, ofreciendo la imagen de sistema único.
  - Esa “transparencia” evitará que los fallos de cualquier subconjunto de ordenadores puedan ser observados por los usuarios.
    - Replicación / Redundancia.

1. Introducción

- ¿Cómo afecta esto a las métricas del tema 1?
  - Obliga a que las aplicaciones se estructuren en múltiples módulos.
  - Exige que haya estrategias de recuperación, con transferencia de estado.
  - Exige que haya múltiples copias de cada componente.
- Todo esto debería favorecer la actualización dinámica.
  - Veamos si es así...

Índice

1. Introducción
2. **Replicación**
3. Revisión histórica
4. Propuestas
5. Resultados de aprendizaje

2. Replicación

- Ya estudiada en RC y SE.
- Aspectos a recordar:
  - Hay varios modelos de replicación.
  - El grado y modelo de replicación dependerá del modelo de fallos asumido en el sistema.
  - Existen protocolos de recuperación...
    - Que transmiten una copia del estado actual a la réplica que se esté recuperando o incorporando.

2. Replicación

- ¿Cómo afecta la replicación (o la distribución) a cada una de estas métricas?
  - Unidad de actualización.
  - Actualización de módulos dependientes.
  - Atomicidad.
  - Transformación de estado.
  - “Type-safety”.
  - Instante de actualización.
  - Actualización de código activo.
  - Diferenciación de código.

2. Replicación

- No afectará a las métricas que no guarden ninguna relación con la arquitectura del sistema.
  - Son éstas:
    - "*Type-safety*".
    - Instante de actualización.
    - Actualización de código activo.
    - Diferenciación de código.
  - Se utilizarán las técnicas vistas en la unidad anterior para solucionar esos aspectos.
- Veamos cómo afecta al resto...

2.1. Unidad de actualización

- Una aplicación distribuida estará formada por múltiples componentes / programas.
  - Todos los componentes colaboran para alcanzar los objetivos de la aplicación.
  - La unidad de actualización será normalmente el módulo o componente.
    - ¡No toda la aplicación!
  - Actualización más sencilla.
    - Los módulos suelen ser pequeños.
    - El acoplamiento debe ser bajo.
    - Las interfaces suelen mantenerse.

2.2. Módulos dependientes

- La actualización de un módulo suele obligar a ubicarlo en otro lugar.
  - Hay que actualizar también las referencias que mantengan los módulos invocadores.
- Ventajas:
  - Toda aplicación distribuida ya proporciona transparencia de ubicación.
    - Mediante stubs o proxies.
    - Indirección (tema 2).
    - Solución: actualizar la referencia de los stubs.

2.3. Atomicidad

- No es aconsejable una actualización atómica.
- Si el módulo está replicado...:
  - Puede actualizarse una réplica mientras las demás siguen prestando servicio.
  - No se incurre en indisponibilidad.
  - No es atómica.
  - ¡Se cumplen los objetivos!!

2.4. Transformación de estado

- Esta métrica comprende dos aspectos:
  - Transferencia de estado.
  - Transformación de estado.
- El segundo dependerá del tipo de actualización efectuado.
  - La distribución o replicación no altera ese aspecto.
    - Deben utilizarse las técnicas descritas en la unidad 2.
- El primero se implanta con los mecanismos de recuperación de réplicas.
  - Están automatizados.
  - Implantan perfectamente la transferencia.

2.4. Transformación de estado

- Esos mecanismos de recuperación suelen implantar la transferencia así:
  - ¿Cómo?: Mediante copia.
  - ¿Qué?: Estado global.
    - Todo el estado “estático” de un determinado componente.
    - Pero no el local / activo.
      - Variables locales de los hilos activos.
  - ¿Cuándo?: Con transferencia ávida.

2. Replicación

- En estas cuatro métricas...
  - El hecho de utilizar un sistema distribuido o un servicio replicado es favorable:
    - Hace más eficiente la actualización dinámica.
    - Mejora la disponibilidad del servicio.
    - No impone sobrecargas que penalicen el rendimiento.

Índice

1. Introducción
2. Replicación
3. **Revisión histórica**
4. Propuestas
5. Resultados de aprendizaje

3. Revisión histórica

- Ampliaremos la revisión de la unidad 2, con algunos trabajos clásicos para sistemas distribuidos:
  - Argus: Tesis de Toby Bloom (1983), resumida en [BD92].
  - PODUS: Trabajos de Segal y Frieder, resumidos en su “survey” [SF93].
  - Tesis de Sameer Ajmani (2004), resumida en [ALS06].
- Presentaremos en la sección 4 propuestas concretas para modelos de replicación.

3.1. Argus (1977-1983)

- Sistema y lenguaje desarrollado en el MIT bajo la dirección de Barbara Liskov.
  - "Guardian"--> Módulo.
  - "Handler" --> Interfaz de un módulo.
- Argus:
  - Lenguaje fuertemente tipado.
    - Sin subtipos. Sin herencia.
  - Pensado para entornos distribuidos.
    - Invocación remota. Vía "handler".
      - Proxy. Indirección.

3.1. Argus (1977-1983)

- Argus:
  - Con un catálogo (servicio de nombres).
    - Permite obtener un “handler” para un “guardian” a partir de su nombre.
    - Utilizado para renovar las referencias de los “handlers” en caso de actualización.
  - “Guardians” replicados por omisión.

3.1. Argus (1977-1983)

- Los mecanismos descritos en [BD92]:
  - No admiten cambios en las interfaces.
    - Salvo adición de operaciones que no afecte a la semántica de la interfaz anterior.
  - Utilizan el soporte de recuperación para lograr la transferencia de estado.
  - Exigen inactividad para llevar a cabo la actualización.
  - La transformación de estado correrá a cargo del programador.

3.1. Argus (1977-1983)

- Los mecanismos descritos en [BD92]:
  - Corrigen automáticamente los módulos dependientes (“handlers”):
    - Se genera una excepción interna.
      - La referencia antigua deja de ser válida si se ha actualizado el “guardian”.
    - Se consulta el catálogo, obteniendo la nueva referencia.
    - Se instala la nueva referencia en el “handler”.
    - Ya no habrá más problemas mientras no haya nuevas actualizaciones.

3.2. PODUS (1989-1993)

- Diseñado y desarrollado por Mark E. Segal y Ophir Frieder.
- Soporte de actualización dinámica que utiliza el procedimiento como unidad de actualización.
  - Necesita MMU con soporte para segmentación.
  - Utiliza compiladores y enlazadores especiales, que incluyen los números de versión en las direcciones lógicas.
- Extendido a sistemas distribuidos, utilizando RPC.
  - Implica indirección.
  - No asume ni necesita replicación.

3.2. PODUS (1989-1993)

- Exige inactividad para realizar la actualización.
  - Pero las unidades de actualización son pequeñas (procedimientos).
    - Es fácil conseguir la inactividad.
    - Se revisan las pilas de los hilos del proceso para verificar si se cumple.
- Soporte para transformación de estado.
  - Funciones de mapeo.
    - Realizan la conversión de datos para que puedan ser utilizados en la nueva versión.

3.2. PODUS (1989-1993)

- También admite cambios en la interfaz del procedimiento.
  - Mediante “hiperprocedimientos”.
    - Son “wrappers” que ofrecen la interfaz antigua.
    - Internamente realizan las transformaciones necesarias para llamar a la versión nueva.
    - También traducirán los resultados antes de retornar el control, si es necesario.

3.3. Ajmani (2002-2006)

- Presenta una propuesta para afrontar la métrica “instante de actualización”.
  - Tradicionalmente se debía elegir entre inmediatez o consistencia.
  - En su tesis garantiza ambas:
    - Actualización “inmediata”.
    - Gestiona multiversionado.
      - Y mecanismos de transformación.
    - Admite “actualizaciones incompatibles”.
      - Cambios en las interfaces.

3.3. Ajmani (2002-2006)

- Objetivos:
  - Mantener versiones antiguas mientras haya clientes o instancias no actualizados.
    - Importante en caso de incompatibilidad de interfaces.
    - Si las interfaces y la semántica se mantienen, la versión antigua puede descartarse de inmediato.
  - Admitir invocaciones para versiones nuevas aunque el objeto todavía no haya cambiado su versión.
    - En algunos casos la transformación de estado será costosa y habrá que hacerla cuando la carga sea muy baja.

3.3. Ajmani (2002-2006)

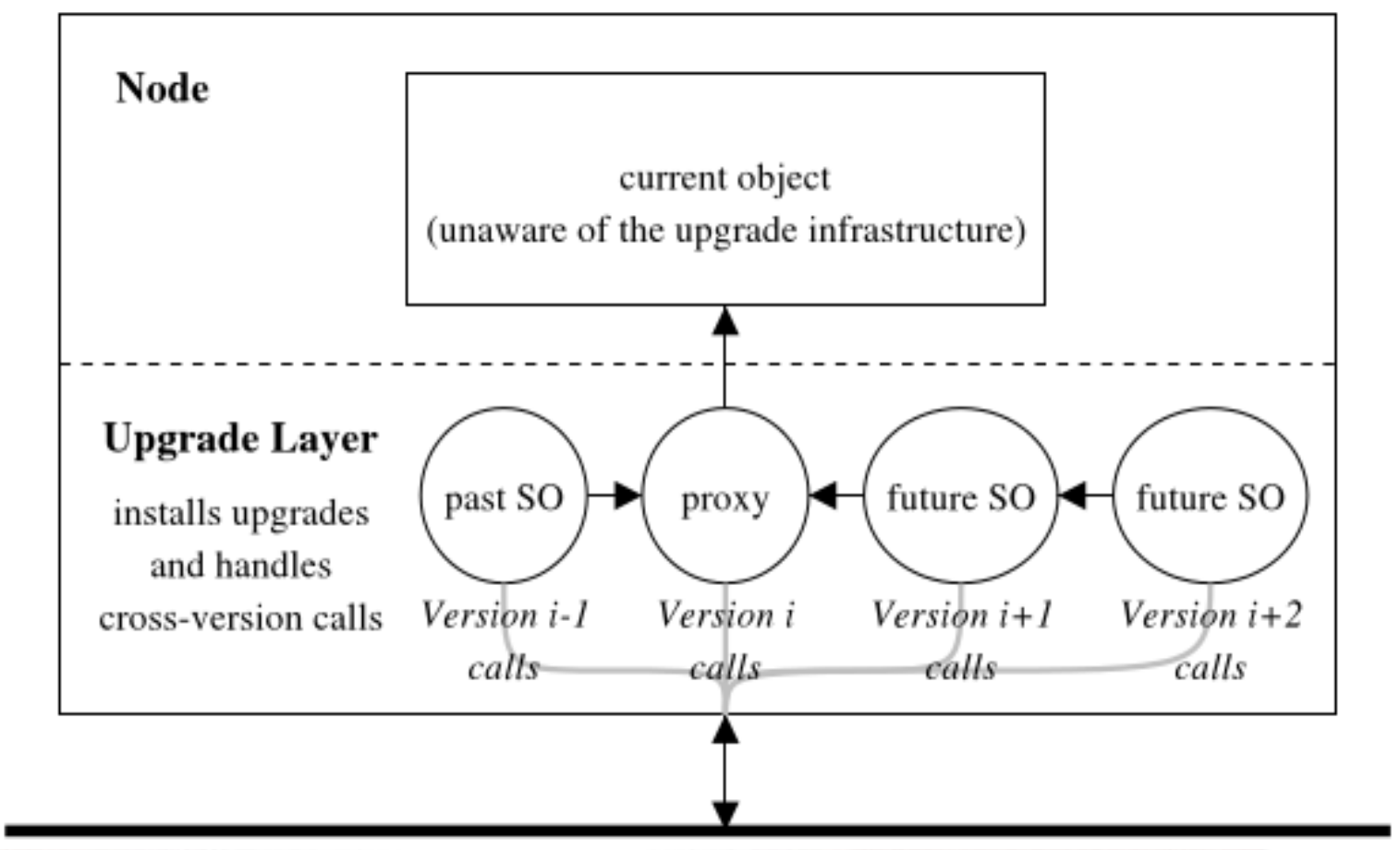
- Sistema distribuido orientado a objetos.
  - No exige replicación.
- Utiliza varias entidades auxiliares:
  - Objeto. Sólo hay una versión activa del objeto en un nodo servidor.
  - Proxy. Es un “esqueleto” o proxy en el dominio servidor.
    - Gestiona las invocaciones a la versión actual del objeto en el nodo local.
  - Objeto de simulación (SO). Ofrece la interfaz de otra versión.
    - Realiza las conversiones necesarias para interactuar con el proxy.

3.3. Ajmani (2002-2006)

- Utiliza varias entidades auxiliares:
  - Funciones de transformación (TF): Realizan la conversión de estado en el objeto cuando realmente cambie su versión. Además:
    - El proxy se transformará en “SO antiguo”.
    - El primer “SO futuro” pasará a ser el “proxy”.
  - Funciones de planificación (SF): Deciden en qué momento se realizará la transformación en cada nodo.
    - Ejemplos: en función de la carga, en función de una planificación global...

3.3. Ajmani (2002-2006)

- Arquitectura de un nodo lógico:



3.3. Ajmani (2002-2006)

- Arquitectura del sistema de actualización:
  - Existe un servidor de actualización (US) para todo el sistema.
    - Centralizado.
    - Mantiene información sobre cada actualización.
      - Qué tipo de cambio se ha realizado y a qué clases de objeto afecta.
      - Necesaria para construir los SO en las capas de actualización (UL).

3.3. Ajmani (2002-2006)

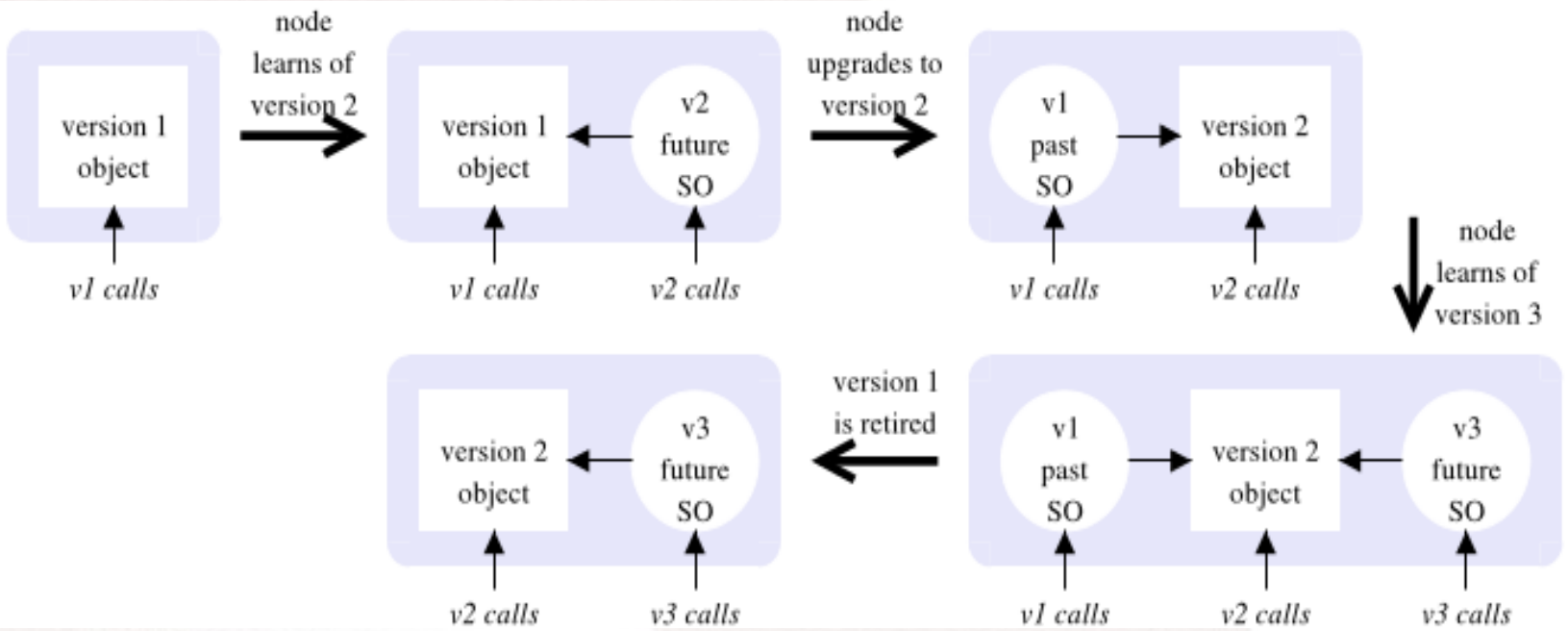
- Arquitectura del sistema de actualización:
  - Hay una base de datos de actualización (UDB).
    - Mantiene información sobre qué versión está siendo utilizada en cada nodo, para cada clase.
    - Todas las invocaciones y respuestas incluyen en sus mensajes el número de versión utilizado.
    - Los servidores pueden dar de baja una versión antigua cuando todas las instancias de esa clase ya estén utilizando una versión superior.
    - ¿Se necesitaría registrar a los clientes?

3.3. Ajmani (2002-2006)

- Arquitectura del sistema de actualización:
  - Cada nodo tiene una capa de actualización (UL).
    - Hay una UL por cada clase de objeto.
    - Mantiene los SO del objeto que gestiona.
    - Ejecuta las SF. En base a ellas:
      - Aplica las TF cuando haya un cambio real de versión.
      - Actualiza su información en la UDB.
      - Si el US observa en la UDB que alguna versión antigua ya no está presente en ningún nodo, informará a los UL para que eliminen el SO antiguo correspondiente.

3.3. Ajmani (2002-2006)

- Ejemplo de transiciones:



3.3. Ajmani (2002-2006)

- Para ejecutar una TF (función de transformación):
  - El UL bloquea todas las invocaciones entrantes.
  - Finalizan todas las invocaciones en curso.
  - Se procede a transformar el estado.
    - En paralelo: Se cambian los proxies y SO, utilizando la información guardada en el US sobre el cambio de versión a aplicar.
  - Se admiten nuevas invocaciones.

Índice

1. Introducción
2. Replicación
3. Revisión histórica
4. **Propuestas**
5. Resultados de aprendizaje

4. Propuestas

- Ninguno de los sistemas estudiados hasta el momento ha aprovechado todas las ventajas que ofrece la replicación a la hora de actualizar.
- Veamos algunas propuestas que sí lo hacen:
  - Solución básica [SM02].
  - Eternal [TMM01].

4.1. Solución básica

- Esta solución básica fue publicada por Solarski y Meling [SM02].
  - Aplicada al modelo de replicación activo.
  - En la tesis doctoral de Solarski, se aplica también al modelo pasivo.
  - Asume:
    - Compatibilidad de interfaces.
    - Compatibilidad de estado.
  - Válida para optimización de rendimiento.
    - También para corrección de errores menores.
    - No reflejados en el estado.

4.1. Solución básica

- Necesita:
  - Recuperación mediante transferencia de estado.
    - La recuperación debe ser coherente con un sistema de comunicación a grupos con sincronía virtual.
      - Para saber qué estado transferir y cuál será el primer mensaje a procesar por la nueva réplica.
  - Para iniciar la actualización de una réplica concreta, ésta no debe tener ninguna invocación “pendiente”.
    - Cola de mensajes de entrada vacía.

4.1. Solución básica

- Algoritmo (modelo activo):
  1. Difundir un mensaje de inicio de actualización a todas las réplicas.
  2. Se elige una réplica candidata:
     1. Se elimina del grupo.
     2. Se activa otra réplica con el nuevo código, ya preparada.
        - Cambio de vista.
     3. Se transfiere el estado desde cualquier otra.
  3. Se sigue iterando mientras queden réplicas con la versión antigua.

4.1. Solución básica

- ¿Por qué funciona?
  - El estado mantenido en ambas versiones no exige una fase de transformación.
  - Sólo se pretende actualizar el código.
  - La sincronización necesaria la proporciona el sistema de comunicación a grupos (SCG).
    - La eliminación de la réplica con la versión antigua y la adición de la réplica con la versión nueva deben solicitarse al SCG.
    - Así se marca la frontera del cambio de vista.
      - La réplica nueva no “pierde” ninguna invocación.
      - Por eso se exigía una cola de entrada vacía en la réplica a actualizar.

4.1. Solución básica

- Algoritmo (modelo pasivo):
  1. Añadir una nueva réplica secundaria con el nuevo código.
  2. Sustituir, una por una, cada réplica secundaria original por otra con la nueva versión.
  3. Eliminar la réplica primaria original.
     - El soporte de replicación elegirá una nueva réplica primaria.
     - Se mantiene el grado de replicación.

4.1. Solución básica

- ¿Por qué funciona?
  - Sigue asumiendo que el estado a mantener es idéntico en ambas versiones.
  - El soporte de replicación realizará todas las transferencias de estado necesarias.
  - Y también el cambio de rol cuando el primario original es eliminado.

4.2. Eternal

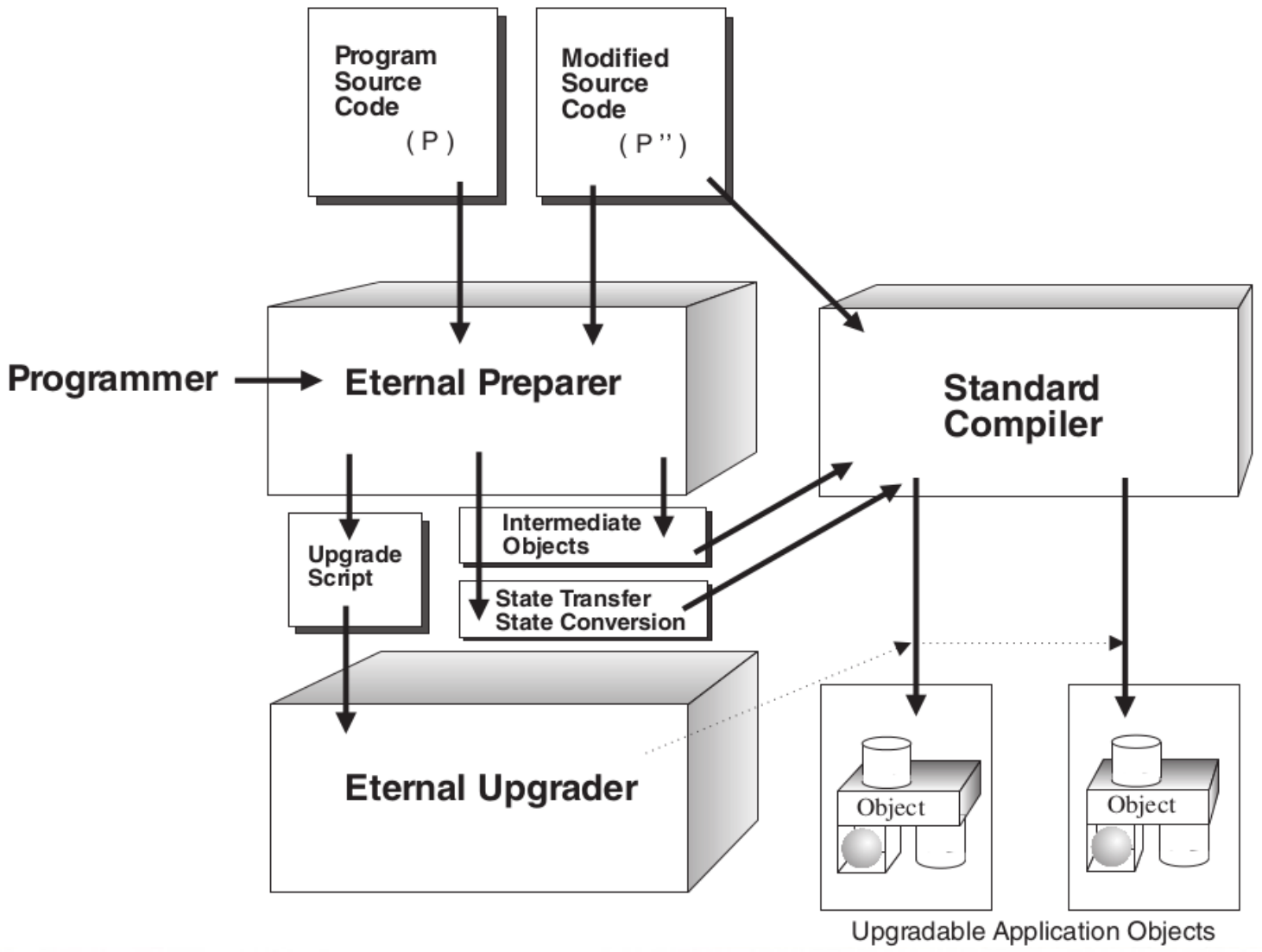
- La solución básica no funcionará en el modelo de replicación activo cuando:
  - Se deba realizar una transformación de estado entre las versiones.
  - Haya alguna diferencia menor en las interfaces.
    - Se exigiría cierta sincronización para aplicar estos cambios / transformaciones “a la vez” en las réplicas.
    - Como no son inmediatos, habría un intervalo de indisponibilidad (largo).
- Eternal resuelve ambos problemas.

4.2. Eternal

- La solución propuesta en el sistema Eternal [TMM01] se basa en un “Gestor de evolución”.
- Este gestor:
  - Comprende dos componentes:
    - Preparador. Compara las dos versiones de código.
      - Métrica 8 del tema 2: Código fuente.
    - Actualizador. Realiza la actualización.
      - Exige inactividad para iniciar la actualización.

4.2. Eternal

- Componentes del gestor de evolución:



4.2.1. Preparador

- ¿Qué hace?
  - Examinar las diferencias entre el código fuente de la versión antigua y la nueva.
  - En base a esas diferencias, genera:
    - El código de conversión de estado.
      - A utilizar en el momento del cambio de versión.
    - Una “clase intermedia”.
      - Superconjunto de las clases antigua y nueva.
      - A utilizar durante la transición.

4.2.1. Preparador

- Genera también una secuencia de “desactualización”.
  - Aplicable en caso de errores en los componentes que ha generado.
- Si hay cambios en las interfaces:
  - El preparador indica en sus resultados al actualizador que:
    - Actualice todas las clases dependientes (objetos clientes).
    - Inactive tanto a los objetos actualizados como a sus dependientes, antes de empezar.

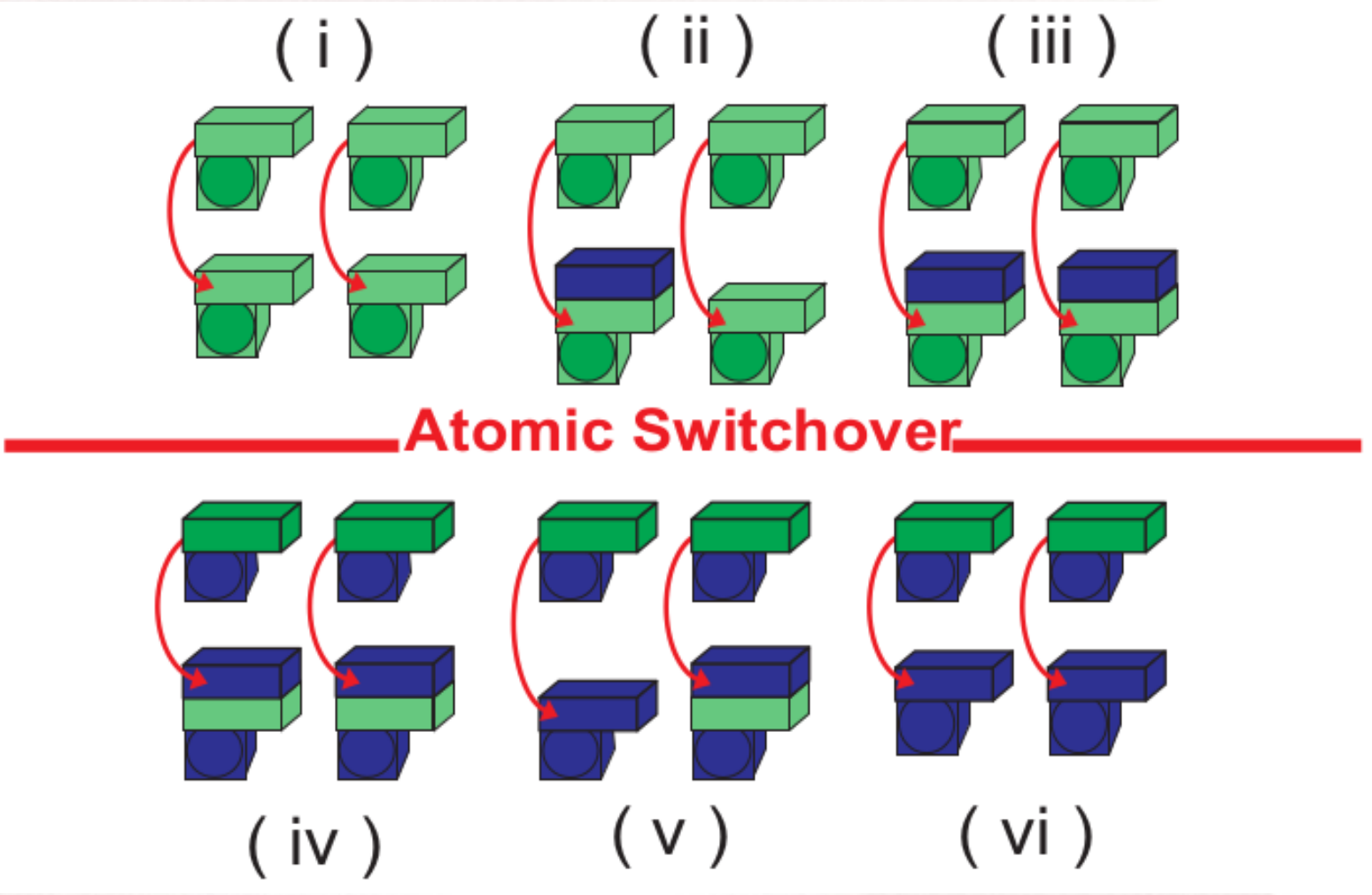
4.2.2. Actualizador

- Este componente realiza los siguientes pasos:
  - Utilizando el protocolo básico de la sección 4.1 (modelo activo):
    - Reemplaza una a una las réplicas de la versión antigua por réplicas de la versión intermedia.
      - Estas réplicas tienen el código de ambas versiones.
      - Siguen recibiendo invocaciones de la versión antigua.
  - Cuando están todas actualizadas, se realiza la transformación de estado.
    - A partir de ese punto, sólo se admiten invocaciones de la versión nueva.
  - Se vuelve a utilizar el protocolo básico de la sección 4.1 para reemplazar cada versión intermedia por la versión nueva.
  - La actualización termina.

4.2.2. Actualizador

- En la próxima hoja:
  - Se ilustra un ejemplo de actualización en un servicio replicado.
    - Dos réplicas.
  - En cada paso:
    - Los dos componentes superiores representan a clientes.
    - Los inferiores, a las réplicas.
    - Color verde: versión inicial.
    - Color azul: versión actualizada.

4.2.2. Actualizador

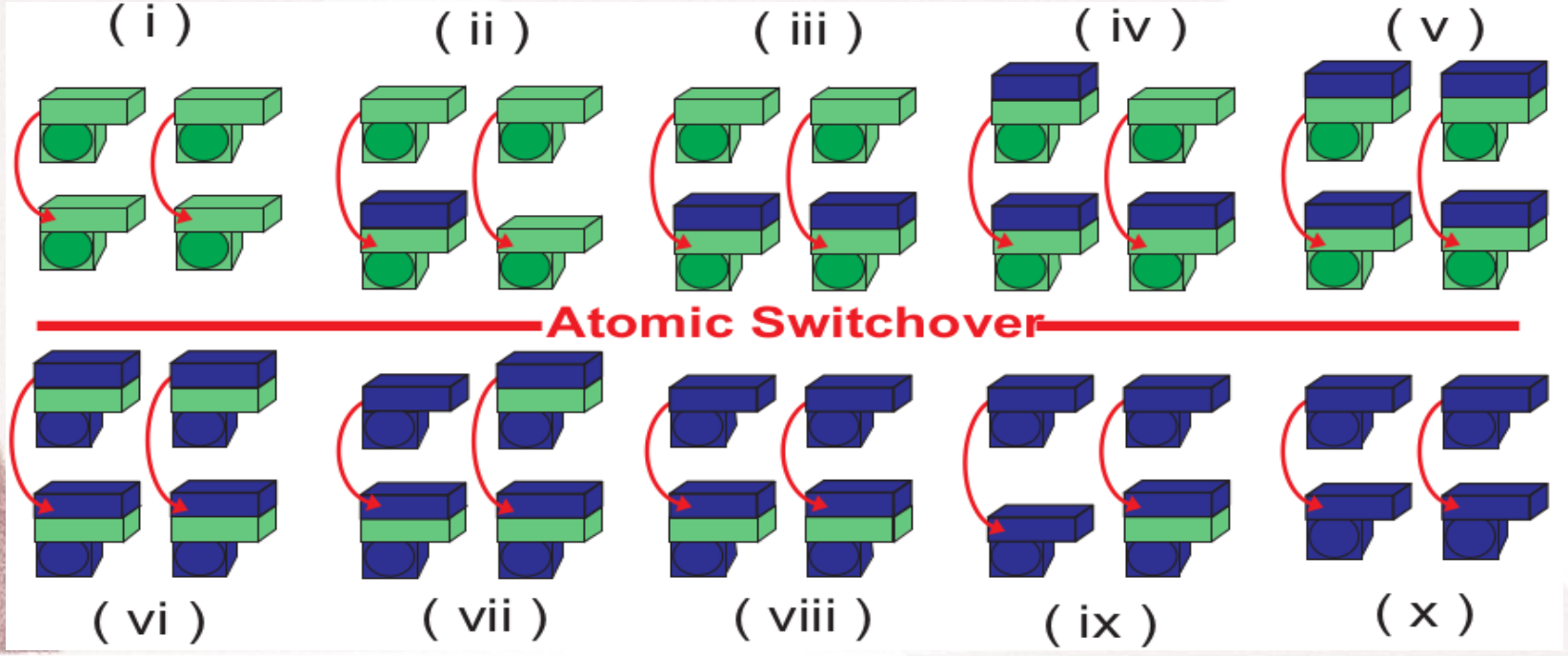


4.2.2. Actualizador

- Pasos:
  i. Estado inicial.
  ii. Actualización intermedia réplica 1.
  iii. Actualización intermedia réplica 2.
  iv. Transformación de estado en clientes y réplicas.
    - A partir de ahora, se invoca la nueva interfaz.
  v. Actualización definitiva réplica 1.
  vi. Actualización definitiva réplica 2.

4.2.2. Actualizador

- Si las interfaces fueran incompatibles...
  - La actualización también tendrá que aplicarse a los clientes.
  - En lugar de seis pasos, se necesitarán diez.



Índice

1. Introducción
2. Replicación
3. Revisión histórica
4. Propuestas
5. **Resultados de aprendizaje**

5. Resultados de aprendizaje

- Al finalizar esta unidad, el alumno debe ser capaz de:
  - Identificar las facilidades que ofrece un sistema de replicación para actualizar dinámicamente un servicio distribuido.
  - Conocer algunas soluciones de actualización dinámica de servicios replicados.
  - Revisar qué problemas quedan pendientes en la actualización de servicios replicados.