7- Acciones Atómicas, Tareas Concurrentes y Fiabilidad

1. Introducción

Las tareas concurrentes son aquellas capaces de cooperar, ejecutándose en paralelo de una forma fiable en presencia de errores en el mismo entorno. Esta interacción presenta una serie de comportamientos:

- **Independencia**: no hay comunicación ni sincronización entre ellas y la recuperación de los errores se realiza de forma aislada.
- **Cooperación**: tienen un propósito común y la comunicación y sincronización entre ellas se realiza de forma regular. La recuperación de errores se realiza de forma conjunta.
- **Competencia**: las tareas no trabajan para un fin compartido, pero necesitan comunicarse y sincronizarse para acceder a recursos compartido. La recuperación de errores queda relacionada con el uso de dichos recursos.

2. Acciones atómicas

La interacción entre dos o más tareas no siempre implica una sola comunicación, y en ocasiones es necesaria la interacción de dos o más tareas. Además, las tareas involucradas deben ver un estado del sistema coherente. Con tareas concurrentes, la interacción entre grupos de tareas es muy sencilla y la actividad conjunta de un grupo de tareas debe ser vista como una acción atómica o indivisible desde fuera.

Se puede definir una **acción atómica** como la actividad conjunta que se desarrolla entre un grupo de tareas para llevar a cabo un fin determinado. Vistas desde fuera, una acción es atómica si las tareas que la llevan a cabo no son conscientes de la existencia de otras tareas activas, y ninguna otra tarea activa es consciente de las actividades de las primeras, durante el tiempo en el que llevan a cabo la acción atómica. Forman un conjunto de tareas indivisibles e instantáneas. De esta forma, estas tareas:

- No tienen comunicación con otras tareas externas.
- No detectan cambios de estado externos.
- Los cambios de estado internos no se comunican hasta finalizar.

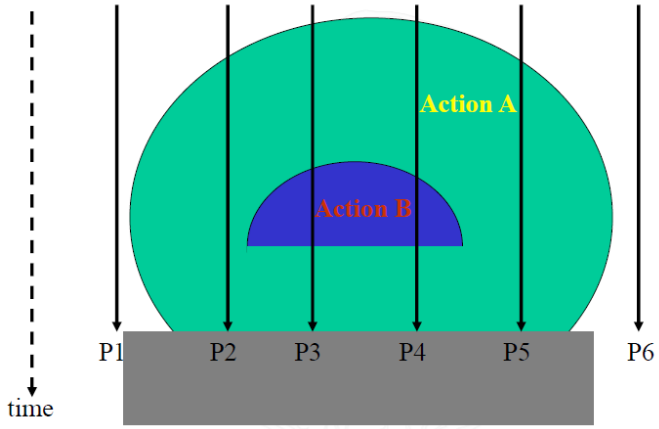
Tipos de acciones atómicas

Que las acciones atómicas sean vistas desde fuera como indivisibles e instantáneas no implica que no pueda haber una interacción entre la acción atómica y el resto del sistema. Estas interacciones no deben tener efecto sobre el resto del sistema y éste no debe recibir información del proceso interno de la acción.

Con esto, se pueden considerar dos formas de llevar a cabo una acción atómica:

- **Acciones atómicas estrictas**: no hay comunicación con el resto del sistema mientras se lleva a cabo la acción.
- **Acciones atómicas no estrictas**: sí existe comunicación con otras tareas externas. Esta comunicación no puede tener efecto en la acción atómica, y el resto del sistema no puede recibir información sobre el proceso de la acción, aunque sí se permite la comunicación con el gestor de recursos.

Aunque las acciones atómicas son indivisibles hacia fuera, pueden tener una estructura interna. Las **acciones atómicas anidadas** permiten la descomposición modular de las acciones de tal forma que las tareas de las acciones anidadas serán un subconjunto de las tareas del nivel exterior y no podrán comunicarse con procesos externos.



Dado que se permite la comunicación hacia el exterior con los gestores de recursos, existen **acciones atómicas de dos fases**, que ayudan a definir una política segura de uso de recursos en dos fases:

- Fase creciente: se realizan las peticiones de recursos que se van a necesitar para realizar la acción.
- Fase decreciente: se produce la liberación de recursos y no puede haber nuevas peticiones de recursos

De esta forma, se asegura así la integridad de la acción atómica. Sin embargo, la liberación temprana de recursos puede hacer más difícil la recuperación de errores ya que, en el momento del error, es posible no disponer de suficientes recursos para poder realizar la recuperación

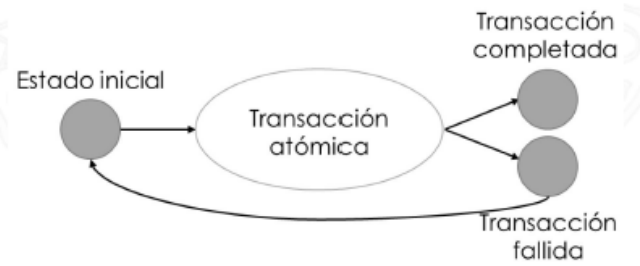
Requisitos de las acciones atómicas

Estos son los requisitos de implementación para que un lenguaje de programación de tiempo real pueda soportarlas:

- **Límites bien definidos**: la acción atómica debe tener definidos:
  - Comienzo y final: son los lugares de cada tarea asociada a la acción en los cuáles ésta tiene su principio o fin
  - Demarcación: separa las tareas involucradas en la acción del resto del sistema.
- **Indivisibilidad (aislamiento)**: implica que no haya intercambio de información entre tareas dentro y fuera de la acción (con excepción del gestor de recursos). No hay sincronización entre las tareas al comienzo de la acción, pero el final de la acción sí implica un proceso de sincronización.
- **Anidamiento**: las acciones atómicas pueden anidarse siempre que no haya solapamiento entre distintas acciones → anidamiento estricto.
- **Concurrencia**: los lenguajes deben permitir ejecutar diferentes acciones atómicas de forma concurrente, aunque por sus características, el efecto de la ejecución concurrente debe ser el mismo que el de la ejecución secuencial.
- Las acciones atómicas deben permitir la programación de procedimientos de recuperación.

Transacciones atómicas

Una **transacción atómica** es un caso particular de las acciones atómicas en las que se puede fallar (no éxito) o tener éxito. Un fallo es la ocurrencia de un error del que la acción no ha podido recuperarse. Para evitar estados inconsistentes, se puede hacer una recuperación de errores hacia atrás de tal manera que los componentes son devueltos a su estado original.



Así, no existe una ejecución parcial de una transacción atómica.

Las propiedades de las transacciones atómicas son:

- **Atomicidad**: los estados intermedios no pueden ser observables.
- **Consistencia**: se garantizan resultados consistentes o se produce una situación de aborto con la que se vuelve al estado inicial.
- **Asilamiento**: no interfiere con otras transacciones.

- **Durabilidad**: los resultados de una transacción pueden ser recuperados ante cualquier tipo de fallo.

No todas las acciones atómicas pueden ser consideradas como transacciones atómicas, diferenciándose estas últimas por:

- **Atomicidad de fallo**: la transacción debe completarse satisfactoriamente o no tener efecto.
- **Atomicidad de sincronización**: la transacción es indivisible y ninguna otra transacción puede observar su ejecución parcial.

Un aspecto importante de las transacciones atómicas es que no se adecuan a los sistemas tolerantes a fallos pues no permiten la recuperación de errores. Siempre se recuperan hacia atrás, volviendo al estado inicial y hay una situación de aborto cuando se produce un error. No permiten otro tipo de recuperaciones de errores y el programador no tiene control sobre este mecanismo (es fijado por el sistema).

3. Acciones atómicas y recuperación de errores

Recuperación de errores hacia atrás

En este caso, si ocurre un error en una acción atómica, las tareas involucradas pueden ser retrotraídos al comienzo de la acción. De esta forma, una vez de vuelta al comienzo de la acción, se pueden ejecutar algoritmos alternativos. La atomicidad asegura que las tareas no han comunicado valores erróneos a otros procesos exteriores. Todas las tareas deben definir módulos alternativos que se ejecutarán si se produce algún error en alguno de los módulos primarios de cualquier tarea

Este mecanismo se basa en el uso de **conversaciones**. Al entrar un proceso en una conversación, se guarda su estado y sólo se permite la comunicación con otros procesos activos en la conversación o con los gestores de recursos. Para abandonar la conversación todos los procesos deben pasar un **test de aceptación**.

Si alguno falla el test, todos los procesos que estén en la conversación deben recuperar el estado guardado e iniciar un método alternativo, aunque se permite llevar a cabo la acción faltando algún proceso involucrado. El éxito dependerá así de que el resto de procesos quieran comunicarse con él. Si alguno de los procesos quiere comunicarse con el que no está activo, existen dos opciones:

- Bloqueo y espera.
- Continuar sin la comunicación.

La principal ventaja del uso de conversaciones es que es posible especificar conversaciones donde la participación no sea obligatoria (no todos los procesos o tareas tienen que formar parte de la conversación) y que los procesos con plazos de tiempo pueden abandonar la conversación. Sin embargo, cuando falla la conversación, todos los procesos deben realizar módulos alternativos (han de tenerlos definidos), forzando a repetir comunicaciones entre el mismo grupo de procesos (con posibilidades de fallar de nuevo la conversación) y además no es posible modificar el grupo de procesos que interactúan ni el test de aceptación.

Alternativamente, se puede plantear la recuperación de errores mediante el uso de diálogos y coloquios, un diseño más diverso que las conversaciones. Los **diálogos** encapsulan un conjunto de procesos de la misma acción atómica, pero sin proporcionar métodos de recuperación, sólo restauran puntos de verificación y señalan el fallo en el coloquio que lo envuelve. El **coloquio** es un conjunto de diálogos controlando su ejecución y decide las acciones de recuperación si hay fallo, proporcionando un medio para construir alternativas usando un conjunto de procesos que podría ser diferente.

Recuperación de errores hacia adelante

Esta forma de recuperación incluye el **uso de gestores o manejadores de excepciones** para evitar volver a un estado inicial. Si ocurre una excepción en alguno de los procesos activos de una acción atómica, ésta se genera para todos los procesos activos, por lo que cada uno de los procesos puede incluir un gestor o manejador de excepciones para la excepción producida. En este sentido, se puede aplicar el modelo de terminación o el de reanudación:

- **Modelo de terminación**: todos los procesos deben incluir gestores.
- **Modelo de reanudación**: se retoma la actividad en el punto en el que ocurrió la excepción.

En cualquier de los dos modelos, puede producirse la excepción estándar `atomic_action_failure` cuando no existe un manejador en ninguno de los procesos activos o falla uno de ellos. Esta excepción indica un fallo en la acción atómica y se generará para todos los procesos.

```
Accion
      <cuerpo de la accion>
except when
      <Identificador de excepcion 1>: <manejador>;
      ...
      <Identificador de excepcion N>: <manejador>;
else
      atomic_action_failure
      <modulo alternativo>
else
      error
end
```

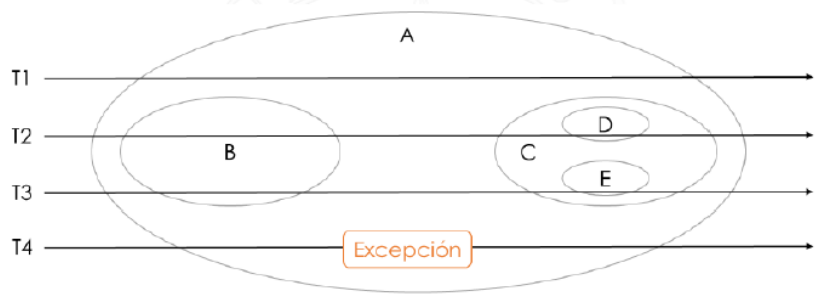
Excepciones concurrentes

Hacen referencia a aquellas excepciones que se pueden generar al mismo tiempo en más de una tarea. En esta situación puede haber manejadores distintos en cada proceso, por lo que puede hacerse difícil elegir manejador. Además, puede darse la existencia de una tercera excepción dada por la conjunción de dos o más excepciones.

Una estrategia muy extendida para tratar este tipo de excepciones es hacer uso de un árbol de excepciones, donde se elige la excepción raíz del subárbol más pequeño que las contenga a todas. Cada proceso puede tener un árbol distinto de excepciones. Si las excepciones son compartidas entre los distintos procesos, se da una situación más sencilla para la elección de la excepción.

Excepciones en acciones atómicas anidadas

En este caso, todas las tareas involucradas en la acción deben participar en la recuperación, pue ésta incluye a la acción interna, que es indivisible.



Una posible solución para estas excepciones es retener la generación de la excepción hasta que finalice la acción interna, aunque puede traer problemas si existen tiempos límite ya que la excepción puede estar relacionada con la acción interna. Otra solución estará en permitir abortar las acciones internas con mecanismos de tolerancia a fallos en la acción interna para abortarse. En este caso, si no es posible abortar se daría una excepción `atomic_action_failure`.

4. Notificación asíncrona

La mayoría de los lenguajes de programación (Java entre ellos) soportan mecanismos de notificación asíncrona para la recuperación de errores, permitiendo que una tarea llame la atención de otra sin esperas.

Hay dos modelos distintos para ello:

- **Reanudación** (manejo de eventos): se comporta como una interrupción software, donde cada proceso indica las interrupciones que manejará y cuando recibir cada evento. El manejador responde al evento asíncrono y tras manejarlo, el proceso se reanuda. Es decir, el flujo de control del proceso cambiará sólo temporalmente, hasta que se trate el evento, siguiendo luego el proceso su ejecución. Es posible asociar un hilo distinto con el evento
- **Determinación** (transferencia asíncrona de control): cada proceso especifica un dominio de ejecución en el que podrá recibir notificaciones asíncronas que lo finalizarán. Si una petición de transferencia asíncrona de control ocurre dentro de un proceso, pero no dentro de su dominio de ejecución, el proceso no recibe la notificación y será ignorada o encolada

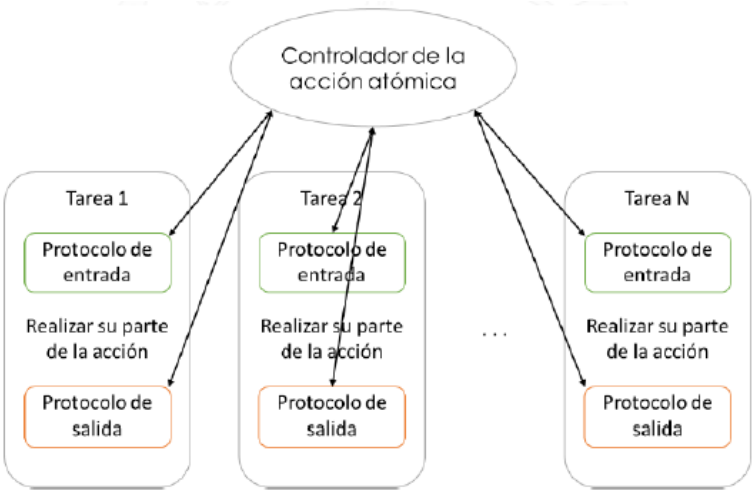
El requisito fundamental por el que se utiliza la notificación asíncrona es porque permite que un proceso responda rápidamente a una condición detectada por otro proceso. Hay ocasiones en que la notificación síncrona (esperas o sondeos) no es adecuada:

- Recuperación de errores
- Cambios de modo de operación
- Planificación utilizando computaciones parciales/imprecisas
- Interrupciones de usuario (entornos interactivos)

5. Acciones atómicas en Java

Las acciones atómicas se implementan mediante interfaces con las tareas que se quieran implementar en dicha acción atómica. La clase que implementa la interfaz implementará también el controlador, que se encarga de la comunicación entre las distintas tareas.

Este modelo basado en el uso de interfaces permite introducir nuevas tareas de forma muy sencilla. Se crearía una nueva interfaz que herede de la primera y se añadirían los métodos correspondientes a las nuevas tareas que se quieran añadir. Posteriormente, se crean mediante mecanismos de herencia una clase que herede de la clase anterior y que realice las redefiniciones necesarias de los métodos para incorporar las nuevas tareas.




```
public interface accion_atomica_N {
      public void tarea1();
      public void tarea2();
      ...
      public void tareaN();
}
```

```
public class control_accion_atomica implements accion_atomica_N{
   protected controlador control;

   public control_accion_atomica(){ // Constructor
      control = new controlador();
   }

   class controlador{ // Clase interna que define el controlador
      protected boolean tarea1_inicio, tarea2_inicio, …, tareaN_inicio;
      protected int tareas_terminadas;
      protected int numero_tareas;
      protected int tareas_salientes;

      controlador(){
         tarea1_inicio = false;
         tarea2_inicio = false;
         …
         tareaN_inicio = false;
         tareas_terminadas = 0;
         numero_tareas = N;
         tareas_salientes = N;
      }

      synchronized void ejecutar1() throws InterruptedException{
         while (tarea1_inicio) wait();
         tarea1_inicio = true;
      }

      synchronized void ejecutar2() throws InterruptedException{
         while (tarea2_inicio) wait();
         tarea2_inicio = true;
      }

      …

      synchronized void ejecutarN() throws InterruptedException{
         while (tareaN_inicio) wait();
         tareaN_inicio = true;
      }

      synchronized void terminar_tarea() throws InterruptedException{
         tareas_terminadas++;
         if (tareas_terminadas == numero_tareas){
            notifyAll();
         }
         else while (tareas_terminadas != numero_tareas){
            wait();
            tareas_salientes--;
            if(tareas_salientes == 0){
               tarea1_inicio = false;
               tarea2_inicio = false;
               …
               tareaN_inicio = false;
               tareas_realizadas = 0;
               tareas_salientes = N;
               notifyAll();
            }
         }
      }
   } // Fin de la clase interna del controlador
```

```
public void tarea1(){
   boolean hecho = false;
   while (!hecho){
      try{
         control.ejecutar1();
         hecho = true;
      }
      catch(InterruptedException e){ // Ignorar
      }
   }
   // Realización de la tarea
   hecho = false;
   while (!hecho){
      try{
         control.terminar_tarea();
         hecho = true;
      }
      catch(InterruptedException e){ // Ignorar
      }
   }
}
```

6. Notificación asíncrona en Java

Para la notificación asíncrona en Java, Java soporta ambos modelos de notificación, ambos en la especificación Java Real-Time:

- **Reanudación**: Java proporciona entidades para el manejo de eventos asíncronos donde los manejadores son entidades programadas, no interrupciones.
- **Terminación**: Java incorpora la transferencia asíncrona de control (ATC) y el manejo de excepciones asíncronas para tiempo real.

Manejo de eventos asíncronos en Java

Para ello, se proporcionan tanto eventos asíncronos (AE) como manejadores asociados (AEH). Un manejador puede ser asociado a uno o más eventos, y un evento puede tener más de un manejador asociado. Además, cada manejador tiene un contador del número de disparos pendientes (cuando un evento asociado es disparado, el contador se incrementa). Existen tres clases principales asociadas con Aes:

- `AsyncEvent`: representa al propio evento.
- `AsyncEventHandler` representa al manejador de eventos asíncronos.
- `BoundAsyncEventHandler`: implementa los manejadores dedicados.

```
public class AsyncEvent
{
  public AsyncEvent();

  public void addHandler(AsyncEventHandler handler);
  public void removeHandler(AsyncEventHandler handler);
  public void setHandler(AsyncEventHandler handler);

  public void bindTo(java.lang.String happening);
  // bind to external event

  public void fire();
  // Execute the run() methods of the set of handlers
  ...
}
```

```
public class AsyncEventHandler implements Schedulable
{
  //constructors
  public AsyncEventHandler();
  public AsyncEventHandler(java.lang.Runnable logic);
  public AsyncEventHandler(
    SchedulingParameters scheduling,
    ReleaseParameters release, MemoryParameters memory,
    MemoryArea area, ProcessingGroupParameters group);
  ... //various other constructors

 //methods needed for handling the associated event
  protected final int getAndClearPendingFireCount();
  protected int getAndDecrementPendingFireCount();
  protected final int getPendingFireCount();

  public void handleAsyncEvent();
  public final void run();
}
```

Transferencia asíncrona de control en Java

Se puede trabajar la transferencia asíncrona de control (ATC) en el paquete estándar Java a partir de la clase `Thread`, que ofrece algunos métodos como:

```
public void interrupt() throws SecurityException;
public boolean isInterrupted();
public boolean interrupted(); // clears bit
```

Cuando un hilo interrumpe a otro, si el hilo interrumpido está bloqueado (`wait`, `sleep`, `join`), se lanza la excepción `InterruptedException`. Si el hilo que se quiere interrumpir está en ejecución, no tiene efecto inmediato, ya que el propio va comprobando con `isInterrupted` si se le ha interrumpido. Este mecanismo no satisface las necesidades de la notificación asíncrona, pues es el hilo tiene que lanzar el método `isInterrumped` para comprobar si ha sido interrumpido.

En cambio, si se trabaja la ATC con la especificación Java Real Time, sí se permite la interrupción inmediata y está integrada en el mecanismo de manejo de excepciones de Java, añadiendo además una extensión de interrupción de hilos. En este contexto, cada método debe indicar si está preparado para permitir una ATC. Necesita 3 actividades:

- Declaración de una excepción asíncrona `AsynchronouslyInterruptedException` (AIE).
- Identificación de los métodos que pueden ser interrumpidos.
- Señalización de una `AsynchronouslyInterruptedException` al hilo.

```
public class InterruptibleService{
  public AsynchronouslyInterruptedException stop =
                AsynchronouslyInterruptedException.getGeneric();
  public boolean Service() throws AsynchronouslyInterruptedException{
    //code interspersed with calls to nonInterruptibleServices
```

```
public InterruptibleService IS = new InterruptibleService();
// code of RT thread, t
if(IS.Service()) { ... }
else { ... };
```

```
// now another RT thread interrupts t:
t.interrupt();
```

- Si se ejecuta un hilo `t` en un método que no tiene AIE en su lista `throws`, o en una sección diferida, la excepción se marca como pendiente y se tendrá en cuenta cuando `t` esté en un método que sí tenga AIE contemplada.
- Si se ejecuta un hilo `t` en método con AIE en su lista `throws`, si además se ejecuta en un bloque `try`, el control pasa al bloque `catch` y/o se propaga. Si no hay bloque `try`, se propaga directamente.
- Si el hilo `t` está bloqueado mediante `sleep`, `wait` o `join`, `t` debe replanificarse y la excepción se propaga o se marca pendiente, dependiendo del método que produjo el bloqueo del hilo.

En cualquiera de los casos, una vez que la AIE es recibida por el manejador, es necesario comprobar si la ATC es la esperada, para lo que se usa el método `clear` en la clase `AsynchronouslyInterruptedException`. Si no lo es, seguirá pendiente y se propagará a un nivel superior.

```
public void useService(){
  stopNow = AIE.getGeneric();
  try {
    // code with calls to InterruptibleService
  }
  catch (AIE AI) {
    // handle the ATC
    if(!stopNow.clear()) {
       // not my ATC, it must be at a higher level
    }
  }
}
```

Además, en Java se proporciona la interfaz `Interruptible` para facilitar el uso estructurado de la ATC. Esta interfaz se implementa en aquellos objetos que quieran proporcionar un método interrumpible en el cual el método `run` es interrumpible y si es interrumpido, el sistema llama al método `interruptAction`.

```
public interface Interruptible{
  public void interruptAction (
         AsynchronouslyInterruptedException exception);

  public void run (
         AsynchronouslyInterruptedException exception)
         throws AsynchronouslyInterruptedException;
}
```

En este contexto, una vez implementada la interfaz `Interruptible`, se pasa un objeto al método `doInterruptible` de la clase `AsynchronouslyInterruptedException`. De esta forma, el método podrá ser interrumpido invocando al método `fire` de la clase `AsynchronouslyInterruptedException`. Si se quiere tener más control sobre AIE, se pueden utilizar los métodos `disable`, `enable`, `isEnabled` (`disable` difiere la AIE hasta que se invoque `enable`).