Programación Concurrente y Distribuida (PCD)

Programación concurrente

Tema 1. Conceptos fundamentales

En los años 60 surgieron los primeros sistemas operativos. El **Mainframe** implementaba concurrencia a nivel ensamblador (en el SO), a lo que luego se sumaron IBM, General Electric, HoneyWell, Siemens y Telefunken.
En 1972, apareció Concurrent Pascal, que abrió la puerta a otros lenguajes de alto nivel que incorporaban la concurrencia.
Esto fue el auge de la programación concurrente:

- Concepto de **thread** o hilo.
- Lenguajes como Java, dando soporte directo con primitivas específicas.
- Aparición de Internet y de procesadores multinúcleo.

Proceso VS Programa

Un programa es algo estático, que está en el disco duro.
Un proceso es dinámico y consume recursos. Es una actividad asíncrona susceptible de ser asignada por un procesador. Un proceso es concurrente si alguna instrucción se mezcla.

- Un proceso puede ser parte de un programa en ejecución.
- Un programa puede dar lugar a uno o varios procesos.

Relación entre procesos

- Nula
- De colaboración
- De competencia

Concurrencia VS Paralelismo

Si dos procesos se ejecutan simultáneamente es programación paralela. La concurrencia es paralelismo potencial, pero son conceptos diferentes.

Beneficios de la programación concurrente

- Aumento de la velocidad de ejecución
- Aprovechamiento de la CPU
- Solución de sistemas inherentemente concurrentes. Sistemas de control, tecnologías web, interfaces de usuario, simulación, SGDB.

Sistema

- Monoprocesador. La **multiprogramación** permite aprovechar ciclos de CPU mientras otros procesos hacen operaciones de entrada/salida, proporcionar un servicio interactivo a múltiples usuarios y dar

una solución adecuada a problemas concurrentes. Todos los procesos comparten la **misma memoria**, por lo que su forma de sincronizarse y comunicarse es mediante el uso de **variables compartidas**.

- Multiprocesador.
    - Sistemas fuertemente acoplados. Tanto procesadores como otros dispositivos (incluida memoria) están conectados a un bus, por lo que **comparten la misma memoria**. Cada procesador puede tener su propia memoria local, pero la comunicación se realiza mediante variables compartidas.
    - Sistemas débilmente acoplados. No hay memoria compartida, sino que cada procesador está conectado con los demás mediante algún tipo de **enlace de comunicación**. Por ejemplo, los sistemas distribuidos están formados por nodos. Cada nodo puede ser a su vez mono o multiprocesador.

Programa

- Concurrente
- Paralelo
- Distribuido [orden de ejecución, indeterminismo]

Creación de threads en Java

- extends Thread
- implements Runnable

Características de los sistemas concurrentes

- Orden parcial de las instrucciones.
- Indeterminismo.

Problemas inherentes a la programación concurrente

- **Exclusión mutua**. Solo uno de los procesos está en la sección crítica en un instante dado.
- **Sección crítica**. La porción de código que debe ejecutarse de forma indivisible.
- **Condición de sincronización**. Un estado en el que un proceso no puede hacer una determinada acción hasta que no cambie su estado.

Corrección de programas concurentes

Problemas de seguridad

Las **propiedades de seguridad** se aseguran de que nada malo va a pasar durante la ejecución del programa.

- **Exclusión mutua**. Hay que garantizar que si un proceso adquiere un recurso, los demás deberán esperar a que sea liberado. De lo contrario, el resultado puede ser imprevisto.
- **Condición de sincronización**. Hay situaciones en las que un proceso debe esperar por la ocurrencia de un evento para poder seguir ejecutándose. Hay que garantizar que el proceso no prosigue hasta que no se produce el evento.
- **Interbloqueo** (**deadlock**). Se produce interbloqueo cuando todos los procesos están esperando por un evento que nunca se producirá. También se conoce con el nombre de **abrazo mortal**.

Problemas de vivacidaz

Las **propiedades de viveza** se aseguran de que algo bueno pasará eventualmente durante la ejecución del programa.

- **Interbloqueo activo (livelock)**. Se produce al ejecutar una serie de instrucciones sin hacer ningún progreso. Es como cuando vas caminando por la calle y te apartas a un lado porque viene una persona de frente, pero esa persona se aparta hacia el mismo lado que tú; entonces ambos os apartáis hacia el otro lado, y así hasta que finalmente os ponéis de acuerdo para pasar.
- **Inanición (starvation)**. El sistema en su conjunto hace progresos, pero algunos procesos no avanzan porque no se les otorga el tiempo de procesador necesario para avanzar.

Condiciones de Bernstein

1. $L(Si) \cap E(Sj) = \varnothing$
2. $E(Si) \cap L(Sj) = \varnothing$
3. $E(Si) \cap E(Sj) = \varnothing$

Lectura y escritura no se pueden ejecutar concurrentemente. Escritura y escritura, tampoco.

Tema 2. Primeras soluciones

Soluciones a la exclusión mutua

Los protocolos de entrada y de salida son porciones de código que deben cumplir las siguientes condiciones para resolver satisfactoriamente el problema de la exclusión mutua:

- Exclusión mutua.
- Limitación en la espera.
- Progreso en la ejecución.

Es necesaria también la ejecución rápida, porque es código que tenemos que ejecutar frecuentemente.

Los mecanismos que disponemos para implementar los distintos tipos de sincronización son los siguientes:

- Inhibición de las interrupciones.
- Soluciones basadas en variables compartidas.
    - Espera ocupada (busy waiting).
    - Semáforos.
    - Regiones críticas.
    - Regiones críticas condicionales.
    - Monitores.
- Soluciones basadas en el paso de mensajes.
    - Operaciones de paso de mensajes send/receive.
    - Llamadas a procedimientos remotos.
    - Invocaciones remotas.

Soluciones Software

Las únicas operaciones atómicas que se consideran son las instrucciones de bajo nivel para leer y almacenar (L/S) de/en direcciones de memoria. Si dos instrucciones de este tipo se produjeran simultáneamente, el resultado equivaldría a la ejecución secuencial en un orden desconocido.

Algoritmos no eficientes

Primer intento (no exclusión mutua)

```
  process P0
repeat
  /*protocolo de entrada*/
  a) while v=scocupada do;
  b) v: =scocupada;
  /*ejecuta la sección crítica*/
  c) Sección Crítica0;
  /*protocolo de salida*/
  d) v: =sclibre;
  Resto0
forever
```

Esta solución no es adecuada, ya que dos procesos pueden ejecutar el `while` antes de ejecutar `v: =scocupada;`, lo que provocaría que ambo entren en la sección crítica. **No garantiza la exclusión mutua**.

Segundo intento (alternancia)

```
  process P0
repeat
  while turno=1 do;
  Sección Crítica0;
  turno: = 1;
  Resto0
forever
```

Garantiza la exclusión mutua, pero provoca que el derecho de usar la sección crítica sea alternativo entre los procesos. Esta alternancia no satisface la condición de progreso en la ejecución.

Tercer intento (no exclusión mutua)

```
  process P0
repeat
  a) while C1=enSC do;
  b) C0: =enSC;
  c) Sección Crítica0;
  d) C0: =restoproceso;
  Resto0
forever
```

No garantiza la exclusión mutua.

Cuarto intento (espera infinita)

```
  process P0
repeat
  C0: =quiereentrar;
  while C1=quiereentrar do;
  Sección Crítica0;
  C0: =restoproceso;
  Resto0
forever
```

Satisface la exclusión mutua, pero produce un problema de progreso en la ejecución: si ambos procesos indican que desean entrar a la sección crítica, se quedarán en un bucle infinito (livelock).

Quinto intento (cortesía)

```
  process P0
repeat
  C0: =quiereentrar;
  while C1=quiereentrar do
  begin
    C0: =restoproceso;
    (*hacer algo durante
    unos momentos*)
    C0: =quiereentrar
  end;
  Sección Crítica0;
  C0: =restoproceso;
  Resto0
forever
```

Satisface la exclusión mutua y no produce espera ilimitada, porque existe una esperanza de que se salga de esta situación, pero **no se asegura que se acceda a la sección crítica en un tiempo finito**.

Algoritmo de Dekker

```
  process P0
repeat
  C0: =quiereentrar;
  while C1=quiereentrar do
    if turno=1 then
    begin
      C0: =restoproceso;
```

```
      while turno=1 do;
      C0: =quiereentrar
    end;
  Sección Crítica 0;
  turno: =1;
  C0: =restoproceso;
  Resto0
forever
```

Combina la alternancia y la cortesía, satisfaciendo así las tres condiciones.

Algoritmo de Peterson

```
  process P0
repeat
  C0: =quiereentrar;
  turno: =1;
  while (C1=quiereentrar) and (turno=1) do;
  Sección Crítica0;
  C0: =restoproceso;
  Resto0
forever
```

Es una forma más sencilla y elegante de resolver el problema.

Algoritmo de Lamport (algoritmo de la panadería)

Soluciona el problema de la exclusión mutua para n procesos y se puede usar en entornos distribuidos donde un proceso puede acceder a la memoria de otro proceso solo para leer.
Se basa en las tiendas donde, al entrar, cada cliente recibe un número. El cliente con el número más bajo es el primero en ser servido, pero este algoritmo no se puede garantizar que dos procesos no reciban el mismo número.

```
  process Pi
repeat
  C[i]: =cognum;
  numero[i]: =1+max(numero[0],...,numero[n–1]);
  C[i]: =nocognum;
  for j: =0 to n–1 do
    begin
      while (C[j]=cognum) do;
      while ((numero[j]Ç0) and ((numero[i],i)>(numero[j],j))) do;
    end;
  Sección Críticai;
  numero[i]: =0;
  Restoi
forever
```

- Se verifica la exclusión mutua por el segundo while: los siguientes procesos sacarán un número mayor que el actual y no pasarían.
- Se garantiza el progreso en la ejecución según el orden en el que se toma el número.
- Asegura limitación en la espera.

Un inconveniente de este algoritmo es que los números pueden aumentar en la ejecución, lo que puede sobrepasar la capacidad de cualquier tipo de datos; por lo que este algoritmo no debe ser usado en modelos de alto grado de concurrencia.

Soluciones Hardware

Se utilizan instrucciones especializadas que llevan a cabo una serie de acciones de forma indivisible, como leer y escribir, intercambiar el contenido de dos posiciones de memoria, etc.

- exchange
- getAndSet
- getAndAdd
- test&Set
- testAndSet(a, b). a = b, b = true

Instrucción de decremento/incremento

Decrementa/Incrementa en 1 el contenido de m y copia el resultado en r de forma atómica: `subc(r, m)` y `addc(r, m)`.
Si un proceso permanece mucho tiempo en la sección crítica, la variable m va a actualizarse continuamente, lo que puede producir **desbordamiento**: no satisface la condición de espera limitada.

Instrucción de testset

`testset(m)`

- Comprueba el valor de la variable m.
- Si es 0, lo cambia por 1 y devuelve true.
- En otro caso, no cambia el valor de m y devuelve false.

Por sí sola no satisface la condición de espera limitada, pero se puede utilizar en otros algoritmos que sí satisfacen las tres condiciones.

Otras soluciones

Deshabilitación de instrucciones

Conclusiones tema 2

Ninguna construcción software que conocemos solucionan la exclusión mutua y la condición de sincronización, y encima contienen espera ocupada o espera activa: **spinklock** (tratamiento a muy bajo nivel).
Las instrucciones atómicas a nivel hardware tampoco ayudan demasiado.
Son necesarias nuevas primitivas: las **primitivas de sincronización**.

Tema 3. Mecanismos para la sincronización de Threads

Variables atómicas

Hay clases en Java como `AtomicInteger` que implementa operaciones que se ejecutan de forma atómica (indivisible). Están en el paquete `java.util.concurrent.atomic`.

```
public class ContadorAtomico {
  AtomicInteger contador = new AtomicInteger();

  public ContadorAtomico() { contador.set(0); }

  public void inc() { contador.incrementAndGet(); }

  public int get() { return contador.get(); }
}
```

Cerrojos

```
public class Contador {
  int contador;
  Lock l = new ReentrantLock();

  public Contador() { contador = 0; }

  public void inc() {
    l.lock();
    contador++;
    try {
      l.unlock(); }
      finally { l.unlock(); }
  }

  public int get() { return contador; }
}
```

Primitivas propias de Java

Se puede sincronizar el método entero.

```
public synchronized void metodo() { }
```

Podemos sincronizar solo una parte del código sobre la clase que implementa el método.

```

```

```
public void metodo() {
  synchronized(this) { }
}
```

O podemos sincronizar sobre un objeto compartido por varias instancias.

```
public synchronized void metodo() {
  synchronized(object) { }
}
```

Hay que usar variables de tipo `volatile` para que no se omitan condiciones que puedan ser cambiadas a lo largo de la ejecución por otros procesos.

- `wait()`: indicates to the current thread to abandone the mutual exclusion (the lock). The thread goes to the wait set until it is awakened by another thread through notify() or notifyAll().
- `notify()`: An arbitrary thread is selected from the wait set by the scheduler to go to the ready state.
- `notifyAll()`: All the waiting threads in the wait set go to the ready state.

Estos tres métodos deben llamarse en métodos que estén sincronizados para asegurar que funcionen correctamente. Por ejemplo:

```
synchronized void doWhenTrue() {
  while(!condition)
    try {
      wait();
    } catch (InterruptedException e) { }
    //SC
}

synchronized void setTrue() {
  condition = true;
  notify(); //or notifyAll();
}
```

Más sobre Threads en Java

- Llamadas anidadas a monitores.
- Autothreads (se crean y empiezan a correr en su propio constructor).
- Prioridades. Podemos asignar prioridades a los hilos en Java para que tengan preferencia.
- Daemons. Es un hilo que hace tareas de fondo. Tiene la prioridad más baja y un hilo se puede declarar como daemon con `setDaemon(true)` antes de iniciarlo. El recolector de basura de Java es un ejemplo de daemon.

Monitores

Es una variable accesible solo por sus métodos, que permitirán bloquear procesos que no pueden seguir su ejecución dentro del monitor, y desbloquearlos cuando la situación que provocó su bloqueo ya no se dé. Implementa una cola.

- `await()`. Se bloquea.
- `signal()`. Desbloquear y continuar.
- `empty()`

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Buffer {
    final Lock candado = new ReentrantLock(true);
    final Condition lleno = candado.newCondition();
    final Condition vacio = candado.newCondition();
    int numActual;
    int capacidad;

    @Override
    public void poner() throws InterruptedException {
        candado.lock();
        try {
            while (numActual == capacidad) lleno.await();
            numActual++;
            vacio.signal();
        } finally
            candado.unlock();
    }

    @Override
    public void coger() throws InterruptedException {
        candado.lock();
        try {
            while (numActual == 0) vacio.await();
            numActual--;
            lleno.signal();
        } finally
            candado.unlock();
    }
}
```

Hay también un monitor específico para Lectores/Escritores.

```
import java.util.concurrent.locks.*;
public class Buffer {
  ReadWriteLock mutex = new ReentrantReadWriteLock();
  public void escribir (int elemento) {
    mutex.writeLock().lock();
    try {
```

```
      /* s.c. */
    }
    finally
      mutex.writeLock().unlock();
  }
  public int leer() {
    mutex.readLock().lock();
    try {
      /* s.c. */
    }
    finally
      mutex.readLock().unlock();
  }
}
```

Semántica de la operación signal

- **Desbloquear y continuar (DC)**. Se desbloquea a un proceso de la cola de condición, pero se sigue ejecutando el método del proceso desbloqueador. `While not B do c.await();`
- **Retorno forzado (DR)**. Se desbloquea a un proceso de la cola de condición y el desbloqueador sale inmediatamente del monitor, terminando la ejecución del método. Nos aseguramos de que la condición no cambia. `If not B do c.await();`
- **Desbloquear y esperar (DE)**. El desbloqueador cede el monitor al proceso desbloqueado. El desbloqueador se queda en la cola del monitor. Nos aseguramos de que la condición no cambia. `If not B do c.await();`
- **Desbloquear y espera urgente (DU)**. Cada monitor llevará asociada una cola de cortesía. Cuando un proceso hace signal, cede la exclusión mutua al proceso desbloqueado y se queda en la cola de cortesía. `If not B do c.await();`
- El que tiene mayor coste de programación es el retorno forzado (DR).
- Los más ineficientes son DE y DU por los cambios de contexto al hacer signal y abandonar el monitor. En menor medida DC por el while.

Semáforos

Están formados por un contador y una cola de procesos. Sus métodos son:

- `acquire(s)`. Si s>0: s--; Si no: desbloquear proceso.
- `release(s)`. Si hay procesos bloqueados: desbloquear uno. Si no: s++;
- `init(s, valorInicial)`. s = valorInicial.

```
public class SemaforoParaMoteros {
    Semaphore[] tropaMoteros;
    Semaphore mutex;
    Semaphore todosListos;
    int numeroMoteros;
```

```
    int cont;

    SemaforoParaMoteros(int _numeroMoteros) {
        cont = 0;
        numeroMoteros = _numeroMoteros;
        mutex = new Semaphore(1);
        todosListos = new Semaphore(1);

        tropaMoteros = new Semaphore[numeroMoteros];

        for (int i = 0; i < numeroMoteros; i++)
            tropaMoteros[i] = new Semaphore(0);
    }

    public void esperaAlResto(int i) {
        try {
            mutex.acquire();
            cont++;

            if (cont < numeroMoteros) {
                mutex.release();
                tropaMoteros[i].acquire();
            } else {
                mutex.release();
                llegoLaPlantillaCompleta();
            }

        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    private void llegoLaPlantillaCompleta() throws InterruptedException {
        for (int i = 0; i < numeroMoteros; i++) {
            tropaMoteros[i].release();
        }
    }
}
```

1. Es un mecanismo de **bajo nivel**, no estrucutrado, que fácilmente lleva a errores transitorios.
2. No se puede restringir el tipo de operaciones realizadas sobre los recursos.
3. Puede haber fallos humanos si se **olvida** incluir todas las sentencias que hagan referencia a los objetos compartidos en las secciones críticas.
4. Es difícil **identificar** el propósito de un acquire o release de forma aislada, ya que se usan tanto para la exclusión mutua como para la condición de sincronización.
5. El código es muy **disperso**, lo que provoca que sea difícil de mantener.

Los monitores tienen la misma capacidad de expresión que los semáforos: cualquier problema que solucionemos con semáforos también podríamos solucionarlo con monitores.

Estructuras de datos concurrentes

Non-blocking data structures

Cuando las operaciones de inserción y borrado no se puedan hacer inmediatamente, devolverán un valor especial o lanzarán una excepción.

- Queue. Estructura de datos lineal de tipo FIFO. Operaciones básicas: add(), remove(), element()
- Deque. Estructura de datos lineal que permite insertar y eliminar elementos a ambos lados de la estructura. Operaciones básicas: addFirst(), addLast(), removeFirst(), removeLast()

Blocking data structures

Cuando las operaciones de inserción y borrado no se pueden hacer inmediatamente, el hilo será bloqueado hasta que se pueda hacer la operación.

- BlockingQueue (LinkedBlockingQueue)
- BlockingDeque
- ConcurrentMap
- TransferQueue

Las estructuras bloqueantes añaden a las clases anteriores sus propios métodos para poder realizar los bloqueos:

- BlockingQueue: put(), take()
- BlockingDeque: putFirst(), putLast(), takeFirst(), takeLast()

En la clase `SynchronousQueue<E>` cada operación de inserción debe esperar a su correspondiente eliminación y viceversa. No necesita capacidad interna (ni capacidad) y esta cola no admite elementos nulos.

```
public class Contenedor_LinkedBlockingQueue {
    LinkedBlockingQueue<Object> l;

    Contenedor_LinkedBlockingQueue(int capacidad) {
        l = new LinkedBlockingQueue<>(capacidad);
    }

    @Override
    public void poner() throws InterruptedException {
        l.put(new Object());
    }

    @Override
    public void coger() throws InterruptedException {
        l.take();
    }
}
```

Mecanismos avanzados de creación de threads

Executor

Un executor maneja un número (fijo o variable) de hilos. Le enviamos **tareas** para que las asocie con hilos (si están disponibles). Estas tareas deben implementar `Runnable` o `Callable`.

```
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadPoolExecutor;

ThreadPoolExecutor executor;
executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(n);
// executor = (ThreadPoolExecutor) Executors.newCachedThreadPool();

executor.execute(() -> {
    //whatever
});

Future<Integer> numero = executor.submit(() -> {
    int cont = 0;
    for (int i = 0; i<n; i++) {
        cont ++;
    }
    return cont;
});

executor.shutdown();
try {
    executor.awaitTermination(1, TimeUnit.MINUTES);
} catch (InterruptedException e) {
    System.out.println("Error en awaitTermination");
}

Integer num = numero.get();
```

`Callable` permite implementar Threads que devuelven un resultado, que obtendremos con la clase `Future`.

ForkJoin

`ForkJoinPool` es un tipo especial de Executor, con el que se diferencia por el algoritmo de robo de trabajo (work-stealing). Cuando una tarea está esperando a la finalización de las subtareas, busca otras tareas que no se hayan ejecutado y las comienza a ejecutar; aprovechando al máximo el tiempo de ejecución y, por tanto, mejorando su rendimiento.
Implementa la interfaz `ExecutorService` y el algoritmo de robo de trabajo, maneja los hilos e informa del estado de las tareas y su ejecución.

```
If (problem size > default size){
  tasks=divide(task);
  execute(tasks);
} else {
  resolve problem using another
```

```
    (simple) algorithm;
}
```

`ForkJoinTask` es la clase base para las tareas e implementa las operaciones de `fork()` y `join()`. Normalmente usaremos:

- `RecursiveAction` para tareas que no devuelven resultados.
- `RecursiveTask` para tareas que devuelven un resultado.

```
public class RecursivePedido_FJ extends RecursiveTask<List<Pedido>> {
    static Integer iter = 0;
    final int casoTrivial = 10;
    final double precioMinimo = 12;
    List<Pedido> pedidoOriginal, retornoPedido;

    public RecursivePedido_FJ(List<Pedido> original) {
        this.pedidoOriginal = original;
        retornoPedido = new LinkedList<>();
    }

    public List<Pedido> compute() {
        iter++;
        if (pedidoOriginal.size() < casoTrivial) {
            for (Pedido p : pedidoOriginal) {
                if (p.getPrecioPedido() > precioMinimo) {
                    retornoPedido.add(p);
                }
            }
        } else {
            int mitad = pedidoOriginal.size() / 2;
            RecursivePedido_FJ subPedidos1 = new
RecursivePedido_FJ(pedidoOriginal.subList(0, mitad));
            RecursivePedido_FJ subPedidos2 = new
RecursivePedido_FJ(pedidoOriginal.subList(mitad, pedidoOriginal.size()));
            invokeAll(subPedidos1, subPedidos2);
            try {
                retornoPedido.addAll(subPedidos1.get());
                retornoPedido.addAll(subPedidos2.get());
            } catch (InterruptedException | ExecutionException e) {
                e.printStackTrace();
            }
        }
        return retornoPedido;
    }
}

public static void Version6ForkJoin(List<Pedido> lp) {
  List<Pedido> auxListaPedidos = new LinkedList<>(lp);
  RecursivePedido_FJ taskPedidos = new RecursivePedido_FJ(auxListaPedidos);
  ForkJoinPool pool = new ForkJoinPool();
  pool.execute(taskPedidos);
```

```
    pool.shutdown();
    try {
        Traza.traza(ColoresConsola.GREEN_BOLD, 1, "V6-D. FORKJOIN");
        Traza.traza(ColoresConsola.WHITE_BOLD, 1,
                Thread.currentThread() + " - Lista de pedidos que superan el
precio de " + taskPedidos.getPrecioMinimo() + ": ");
        taskPedidos.get().stream().parallel().forEach(p ->
Traza.traza(ColoresConsola.CYAN_BOLD, 2, p.printConRetorno()));
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Mecanismos avanzados de sincronización

CyclicBarrier

Permite que un conjunto de threads esperen al resto al llegar a un punto de barrera común. La barrera es cíclica porque **puede reutilizarse** tras la liberación de los threads. Soporta un comando `Runnable` opcional que se ejecuta una vez por cada punto de barrera, después de que llegue el último thread, pero antes de que se liberen todos, lo que resulta útil para actualizar datos compartidos antes de continuar con la ejecución.

```
CyclicBarrier cyclicBarrier = new CyclicBarrier(numeroMoteros, () ->
System.out.println("Procesando moteros..."));

public void esperaAlResto() {
    try {
        cyclicBarrier.await();
    } catch (InterruptedException | BrokenBarrierException e) {
        e.printStackTrace();
    }
}
```

CountDownLatch

Permite a uno o más threads esperar a que se realicen una serie de operaciones que se ejecutan en otros threads. Se inicializa con una cuenta (**count**). El método `await()` bloquea al thread actual hasta que el contador llegue a 0 por las invocaciones a `countDown()`, y todos los threads que estaban bloqueados se liberan inmediatamente.

- La principal diferencia entre `CyclicBarrier` y `CountdownLatch` es que `CyclicBarrier` permite que un conjunto de hilos se esperen entre sí, mientras que `CountdownLatch` permite que uno o más hilos esperen a que un conjunto de tareas se completen.
- El contador de `CountdownLatch` no se puede restablecer una vez llegado a cero, mientras que el contador de `CyclicBarrier` sí se puede restablecer y reutilizar varias veces.

Exchanger

Un punto de sincronización donde threads se agrupan por pares e intercambian elementos entre sí. Sirve para intercambiar objetos entre dos hilos utilizando el método `exchange()`. Es como una forma bidireccional de SynchronousQueue. Pueden ser útiles en aplicaciones de algoritmos genético y pipelines. Solo se pueden intercambiar objetos del mismo tipo.

```
import java.util.concurrent.Exchanger;

public class Main {
  public static void main(String[] args) {
    Exchanger<String> exchanger = new Exchanger<String>();
    new Thread(new FirstThread(exchanger)).start();
    new Thread(new SecondThread(exchanger)).start();
  }
}

class FirstThread implements Runnable {
  private String message;
  private Exchanger<String> exchanger;

  public FirstThread(Exchanger<String> exchanger) {
    this.exchanger = exchanger;
    message = "Hello World";
  }

  public void run() {
    try {
      message = exchanger.exchange(message);
      System.out.println("First Thread received: " + message);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}

class SecondThread implements Runnable {
  private String message;
  private Exchanger<String> exchanger;

  public SecondThread(Exchanger<String> exchanger) {
    this.exchanger = exchanger;
    message = "Goodbye World";
  }

  public void run() {
    try {
      message = exchanger.exchange(message);
      System.out.println("Second Thread received: " + message);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
```

```
  }
}
```

Phaser

Se utiliza cuando varias tareas concurrentes se dividen en pasos: `Phaser` permite sincronizar los hilos al final de cada paso, de tal forma que ningún hilo comience con el siguiente hasta que todos hayan terminado el anterior.

```
import java.util.concurrent.Phaser;

public class Main {
  public static void main(String[] args) {
    int nproc = Runtime.getRuntime().availableProcessors();
    ThreadPoolExecutor ex =
(ThreadPoolExecutor)Executors.newFixedThreadPool(nproc);
    for (int i=0;i<nproc;i++) {
      Task t = new Task (i, p, nproc);
      ex.execute(t);
    }
    ex.shutdown();
    try {
      ex.awaitTermination(1, TimeUnit.DAYS);
    } catch (InterruptedException e) {e.printStackTrace();}
    // This message will be the last one to be printed
    System.out.println ("Fin");
  }
}

class Task implements Runnable {
  private i;
  private Phaser phaser;

  public FirstThread(Phaser phaser, int i) {
    this.i = i;
    this.phaser = phaser;
  }

  public void run() {
    System.out.println("Thread "+i+" - Phase One");
    phaser.arriveAndAwaitAdvance(); // Waiting for all the threads to
finish Phase 1
    System.out.println("Thread "+i+" - Phase Two");
    phaser.arriveAndAwaitAdvance(); // Waiting for all the threads to
finish Phase 2
    System.out.println("Thread "+i+" - Phase Three");
    phaser.arriveAndAwaitAdvance(); // It doesn't wait for the others
  }
}
```

Programación distribuida

Tema 4. Programación reactiva

Lambdas

Una expresión lambda se compone de:

- Listado de parámetros separados por comas y encerrados en paréntesis. Por ejemplo: (a,b).
- El símbolo de flecha hacia la derecha: ->
- Un cuerpo que puede ser un bloque de código encerrado entre llaves o una sola expresión.

```
(int a, int b) -> { System.out.println(a + b); return a + b; };

//Función lambda que inicializa la tramitación de cada pedido
executor.execute(() -> {
    restaurante.tramitarPedido(p, latch);
    try {
        Thread.sleep(0);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
});
```

Interfaces funcionales comunes

- `Predicate`. Su método `test` devuelve un booleano: `boolean test (T t)`-
- `Filter`. Devuelve una lista con todos los elementos que cumplen una determinada condición.
- `Consumer`. Acepta un parámetro genérico y no devuelve nada. Su método es `void accept(T)`.
- `Function`. Recibe un parámetro de un determinado tipo y retorna un resultado de tipo diferente (convierte un dato de un tipo en otro tipo diferente). Su método es `R apply (T)`.

Streams paralelos

Un stream es una **secuencia** de elementos pensado para **cálculos** más que para datos. Consumen de una **fuente** no modificable que proporciona los datos (ficheros, colecciones), y van consumiendo/procesando los nuevos elementos. Puede haber **encadenamiento** de operaciones, cada iteración es **interna** y es fácilmente **paralelizable**, pero son recorribles solo una vez.

Existen dos tipos de operaciones sobre streams: **intermedias** y **terminales**. Las primeras devuelven un stream y se pueden concatenar (es el caso de `filter` y `map`). Las terminales producen un resultado (el caso de `collect`).

Algunas operaciones de streams son:

- filter()
- map()
- skip(n)
- limit(n)

- forEach()
- anyMatch()
- noneMatch()
- allMatch()
- findAny()
- findFirst()
- reduce()
- sorted()
- **iterate**()
- **range**()

```
```

En la medida de lo posible, al trabajar con streams, debemos usar lo que se denominan funciones sin estado o bien que el estado sea inmutable, es decir, no pueda ser cambiado por ningún thread. En nuestro caso add es una función con estado que además puede ser cambiado (se dice que es mutable). Hay que huir de esas situaciones.

- A veces no es mejor usar streams paralelos. Para una pequeña cantidad de datos, el coste adicional por el proceso de paralelización no compensa.
- Algunas operaciones funcionan mejor en un stream paralelo que en uno secuencial, y viceversa.
- Cuidado con el boxing/unboxing.
- El **coste computacional** ha de ser considerado. Con N siendo la cantidad de elementos a procesar y Q el coste aproximado de procesar cada elemento a través del stream, el producto de N * Q da una estimación cualitativa aproximada de este coste. Un valor más alto para Q implica una mejor oportunidad de obtener buen rendimiento cuando se utiliza un stream paralelo.

Observables y Observadores

- La **clase** `Observable` tiene el método `subscribe()`.
- La **interfaz** `Observer` tiene los métodos `onNext()`, `onError()`, `onComplete()` y `onSubscribe()`.

```
// Ejemplos de creación de Observables
Observable<String> source = Observable.just("Uno", "Dos", "Tres");
Observable <Book> b = Observable.fromStream (books.stream());
```

Para que un Observer pueda observar un Observable, debe subscribirse a él invocando el método **subscribe** que recibe por parámetro el objeto Observer.

```
source.subscribe (s->System.out.println ("Recibido: "+s),
  t->t.printStackTrace(),
  ()->System.out.println ("Hecho"));
```

También podemos crear un Observable utilizando una lambda que acepta un emisor Observable:

```
Observable <String> o2 = Observable.create(emitter -> {
  emitter.onNext("uno");
  emitter.onNext("dos");
  emitter.onNext("tres");
  emitter.onComplete();
});

o2.subscribe (s->System.out.println ("Recibido: "+s));
```

COLD OBSERVER

Los suscriptores reciben desde el principio todos los valores. Es como escuchar un CD, elijo lo que quiero oír desde el principio.

```
//COLD OBSERVER
Observable<String> source = Observable.just("Uno", "Dos", "Tres");

source.subscribe(a -> System.out.println("Observador 1: "+a));
source.subscribe(a -> System.out.println("Observador 2: "+a.length()));
```

HOT OBSERVER

Es como la radio: me pierdo lo que haya habido antes de mi subscripción.

```
//HOT OBSERVER
ConnectableObservable<String> source2 = Observable.just("Uno", "Dos",
"Tres").publish();

source2.subscribe(a -> System.out.println("Observador 1: "+a));
source2.subscribe(a -> System.out.println("Observador 2: "+a.length()));
source2.connect(); //Desde aquí se mandan TODOS los datos a la vez a los
observadores

ConnectableObservable<Long> source3 = Observable.interval(1,
TimeUnit.SECONDS).publish();

source3.subscribe(a -> System.out.println("Observador 1-interval: "+a));
source3.subscribe(a -> System.out.println("Observador 2-interval:
"+a.length()));
source3.connect(); //Desde aquí se mandan TODOS los datos a la vez a los
observadores
sleep(3000);
source3.subscribe(a->System.out.println("Observador 3-interval: "+a));
//Este es el que pone la radio tarde y se pierde parte del programa
sleep(3000);
```

En el caso de interval, se crea un hilo aparte.

```
Observable.interval(1, TimeUnit.SECONDS).subscribe(s->System.out.println
(Thread.currentThread()+ " "+s ));
System.out.println (Thread.currentThread()+ "En el main ");
sleep(7000); // paramos el hilo principal
```

Concurrencia en Programación Reactiva

```
//Lanzando cada elemento en un thread
Observable.range(1, 10)
  .flatMap(i -> Observable.just(i)
  .subscribeOn(Schedulers.computation())
  .map(i2->intenseCalculation (i2)))
  .subscribe(i -> System.out.println("Current thread: "+
    Thread.currentThread() +" Recibido: " + i + " "+LocalTime.now()));
sleep (12000);
```

Tipos de Schedulers

- `Schedulers.computation()`. Crea y devuelve un scheduler para cómputos. El número de threads depende del procesador. Se permite un thread por procesador, es la mejor opción para bucles o recursividad.
- `Schedulers.io()`. Crea y devuelve un scheduler para entrada-salida (IO-bound work). Pueden usarse más threads si fuera necesario.
- `Schedulers.newThread()`. Crea y devuelve un scheduler que crea un nuevo thread para cada unidad de trabajo.
- `Schedulers.trampoline()`. Crea y devuelve un scheduler que pone trabajo en la cola del thread actual para ejecutar después de que el trabajo actual se complete.
- `Schedulers.from(java.util.concurrent.Executor executor)`. Convierte un Executor en un nuevo scheduler.

```
public static void main(String[] args) {
  int numberOfThreads = 20;
  ExecutorService executor = Executors.newFixedThreadPool(numberOfThreads);
  Scheduler scheduler = Schedulers.from(executor);
  Observable.just("Alpha", "Beta", "Gamma", "Delta", "Epsilon")
    .subscribeOn(scheduler)
    .doFinally(executor::shutdown)
    .subscribe(System.out::println);
}
```

observeOn

Con `observeOn` se puede cambiar de scheduler. Aplica a la cadena que viene detrás de `observeOn`. Con `subscribeOn` da igual dónde se ponga, con `observeOn` no da igual.

```
Observable.just("long", "longer", "longest")
  .doOnNext(System.out::println)
  .subscribeOn(Schedulers.io())
  .map(String::length)
  .observeOn(Schedulers.computation())
  .filter (i->(i > 6))
  .subscribe (length -> System.out.println("item length "+length));

Observable. just("long", "longer", "longest")
  .doOnNext( i->System.out.println ( "Current Thread: " +
Thread.currentThread()+" "+i))
  .subscribeOn(Schedulers. io())
  .map(String::length)
  .doOnNext( i->System.out.println ( "Current Thread: "
+Thread.currentThread()+" "+i))
  .flatMap(j->Observable. just(j).observeOn(Schedulers. computation())
  .doOnNext( k->System.out.println (Thread. currentThread()+" "+k))
  .filter (i->(i > 6)))
  .subscribe ( length -> System. out.println("Current thread: " +
Thread.currentThread()+" length " +length));
```

Tema 5. Mecanismos de paso de mensaje

Características de los mecanismos de paso de mensaje

¿Identificamos emisor y receptor en el proceso de comunicación?

- **Comunicación directa**.
  - **Simétrica** (ambos están identificados). La ventaja es la seguridad (los procesos están perfectamente identificados), pero modificar el nombre de un proceso implica modificar el programa.
    `SEND (A, message)`. Send a message to process A.
    `RECEIVE (B, message)`. Receive a message from process B.
  - **Asimétrica** (el emisor identifica al receptor).
    `SEND (A, message)`. Send a message to process A.
    `RECEIVE (Id, message)`. Receive a message (in Id the system would indicate the identification of the sender).
- **Comunicación indirecta**.
  `SEND (mailbox, message)`. Send a message to the mailbox
  `RECEIVE (mailbox, message)`. Receive a message from the mailbox
  - **Buzón** (mailbox)
    - 1 a 1
    - 1 a n
    - n a 1 (puerto)
    - n a n
  - **Canales**
    - fuertemente tipados
    - comunicación síncrona

    - 1 a 1

Tipo de comunicación

- Síncrona (rendezvous y extended rendezvous)
- Asíncrona
- Paso de mensajes futuro (Future message passing)

Canal (o medio) de comunicación

- **Flujo**
    - unidireccional (Sockets UDP, comunicación asíncrona)
    - bidireccional (Sockets TCP, comunicación RPC)
- **Capacidad** del canal
    - cero (comunicación síncrona)
    - finito (comunicación asíncrona)*
    - infinito (comunicación asíncrona)*
      *buffer asociado al canal
- **Longitud** de los mensajes
    - fija (fragmentación provoca desorden)
    - variable (message = head + body)
- Canales con o sin **tipo**.
- **Parámetros** pasados
    - por referencia (siempre en sistemas distribuidos)
    - por copia (implica compartición de memoria, por lo que nunca se usa en distribuida; menos seguro en concurrencia, pero más eficiente)
- Transmisión de **errores**.

Paso de mensaje síncrono

Se produce una cita, rendezvous (aka synchronous message passing): la comunicación no empieza hasta que ambos, emisor y receptor, estén preparados.

Espera selectiva

Select sentence: Dijkstra, 1975

```
select
RECEIVE (process1, message);
sentences;
or
RECEIVE (process2, message);
sentences
or
...
or
RECEIVE (process3, message);
sentences;
end select;
```

Selective waiting with guards

```
...
select
when condition1 =>
RECEIVE(process1, message);
sentences;
or
when condition2 =>
RECEIVE(process2, message);
sentences;
or
...
or
when conditionN =>
RECEIVE(processN, message);
sentences;
end select;
...
```

Las condiciones solo se evalúan cuando llegan a SELECT.

Paso de mensaje asíncrono

El emisor envía los datos cuando él quiera y el receptor los recibe cuando le venga en gana.

Invocación remota aka extended-rendezvous

La comunicación Extended-Rendezvous es una extensión de la comunicación Rendezvous que permite que más de dos procesos se comuniquen entre sí. En este tipo de comunicación, el emisor espera la recepción del mensaje por parte del receptor y una respuesta determinada.

Remote invocation

El término remoto se refiere a otro **proceso**. El cliente y el servidor pueden estar en el mismo contexto y máquina.

RPC (Remote Procedure Call)

El término remoto se refiere a otro **procesador**. El cliente y el servidor están en distintas máquinas (distribuidas). Por ejemplo: Java RMI y gRPC.

Ambos son modelos de comunicación síncronos, su flujo es bidireccional y son buenos para aplicaciones del estilo cliente/servidor.

Sentencia SELECT

Communication alternatives are modelled using `Selectable` objects. `MailBox`, `Channel` and `EntryPoint` classes inherits from `Selectable`.
The mode of use is as follows. After creating an object of type `Selector`, you attach to it objects of type `Selectable` and finally invokes any of the various selection methods available.
To add the different communication alternatives to a select, we will use the `addSelectable (Selectable s, boolean sender)` method that will allow us, thanks to the sender parameter, to indicate if the added alternative will be used to send (true) or to receive messages (false). Once this is done, before invoking any of the selection methods, we can assign a boolean value to the guard that each communication alternative has associated. If we do not do that, it is assumed that the value assigned is true.

Tema 6. Programación con Sockets

Concepto de sockets

Un socket es un objeto de software que permite **enviar y recibir datos** entre hosts remotos o entre procesos locales. Es un punto de conexión de comunicaciones que posee un nombre y dirección en una red. Los sockets se crean y se utilizan con un sistema de peticiones o de llamadas de función a veces llamados interfaz de programación de aplicación de sockets (API).
Un socket lo crea el sistema operativo. Java implementa sus clases por encima de las funciones que ofrecen el sistema operativo.

- Un **cliente** inicia la comunicación y solicita un servicio al servidor.
- El **servidor** espera peticiones y ofrece el servicio pedido.
- Un **puerto** es una dirección lógica (un número) por el que un servicio se ofrece/accede. Lo proporciona el Sistema Operativo.

Un socket es una abstracción del SO. Las aplicaciones los crean, los usan y los cierran cuando ya no hacen falta. Su comportamiento lo controla el SO.
Los procesos envían/reciben mensajes a través de sockets. Internet se comunica usando sockets. El proceso que se va a comunicar se identifica por su socket. Cada socket tiene un identificador, que está formado por la IP del ordenador y un número de puerto.

Protocolo TCP/IP

Un socket API (Application Program Interface) permite a la aplicación usar los protocolos TCP/IP y define las operaciones permitidas (abrir, leer, escribir, cerrar, etc.) y sus parámetros.

Se transportan **bytes**. La aplicación protocolo proporciona la semántica.

Data marshalling (serialización)

El marshalling o serialización es el proceso de **transformar la representación de memoria** de un objeto en un formato de datos adecuado para el almacenamiento o la transmisión. Se utiliza típicamente cuando los datos deben moverse entre diferentes partes de un programa de computadora o de un programa a otro.

En resumen, el marshalling es el proceso de convertir objetos en flujos de bytes capaces de ser almacenados en dispositivos, bases de datos o de ser enviados a través de la red y, posteriormente, ser capaces de reconstruirlos en los equipos donde sea necesario.

Tipos de sockets

Desde el punto de vista de su comportamiento:

- **Activos**. Pueden enviar y recibir datos a través de una conexión.
- **Pasivos**. Esperan a intentos de conexión. Cuando llega una petición, se asigna un socket activo.

Desde el punto de vista del protocolo:

- Sockets TCP. Stream.
- Sockets UDP. Datagrama.

Sockets TCP (Stream)

Se comportan como una llamada telefónica.

1. El equipo local solicita el establecimiento de comunicación al canal del equipo remoto.
2. Una vez se ha creado el canal, la comunicación puede empezar.

- Se garantiza la llegada.
- Stream de **bytes**: llegada en orden.
- Orientado a **conexión**: un socket por conexión (cliente)
- La conexión se establece y luego los datos se intercambian.

Clases en Java TCP

- `Socket`. Establecimiento del cliente y conexión con servidor.
- `ServerSocket`. Establecimiento de un socket pasivo en un servidor. Se usa para escuchar peticiones.

Sockets UDP (Datagram)

Se comportan de forma similar a una carta postal. Lo más importante es la **velocidad**. Los mensajes se pueden perder o llegar fuera de orden.
Son menos fiables que los TCP, por lo que habría que implementar, si fuera necesario, sus propios mecanismos de verificación.

- Un **socket** para recibir mensajes (como un buzón). No se garantiza la entrega ni el orden.
- Un **datagrama** es un paquete independiente.
- Muchas **direcciones** cada paquete.

Se envían datagramas sin conexión, de forma rápida y sin asentimiento. La comunicación es rápida y simple.

Clases en Java UDP

- `DatagramPacket`. Implementa el `Paquete` de datos que se envían por la red.
    - En la recepción se debe especificar:
        - Un **buffer** donde almacenar los datos recibidos (array de bytes).
        - Un entero indicando el tamaño máximo de los datos a recibir. `DatagramPacket dp=new DatagramPacket(buffer, tam);`
    - En el envío se debe especificar:
        - El buffer de datos a enviar.

        - El tamaño de los datos a enviar.
        - Dirección (address).
        - Puerto.

          `DatagramPacket dp=new DatagramPacket(buffer,tam,direcc,puerto);`
- `DatagramSocket`. Maneja sockets UDP, permitiendo enviar y recibir datos (datagrams) por la red.
    - En la recepción se ha de especificar el puerto en el que las peticiones serán escuchadas.
      `DatagramSocket ds_recep=new DatagramSocket(1234);`
    - En el envío no es necesario especificar parámetros, porque ya están en el mismo datagrama.
      `DatagramSocket ds_envi=new DatagramSocket();`

¿TCP o UDP?

Depende de la aplicación.

- **TCP** cuando se necesite integridad de la información u orden en los mensajes. Por ejemplo en control remoto o transferencia de archivos.
- **UDP** cuando se necesiten comunicaciones rápidas o no importe la pérdida de paquetes. Por ejemplo en aplicaciones de tiempo real (videoconferencias) o aplicaciones distribuidas en una LAN.

Tema 7. Problemas de programación distribuida

Los sistemas distribuidos son programas que se ejecutan en máquinas diferentes.

Programación distribuida: requisitos y posibilidades

- **Mecanismos de comunicació**n, que pueden ser directos o indirectos, y simétricos o asimétricos.
- **Sincronización** asíncrona o síncrona (simple/extended rendezvous).
- Características del **canal**: tipado del canal, parámetros, capacidad y dirección del flujo.

Problemas de la programación distribuida

- Sincronización del reloj.
- Exclusión mutua.
- Detección de terminación.
- Detección de interbloqueo pasivo (deadlock).
- Estado global del sistema.
- Elección de líder.
- Consenso (blockchain).

Algoritmos distribuidos

Algoritmos de sincronización de reloj

- Relojes físicos: Algoritmo de Christian.
- Relojes lógicos: Algoritmo de Lamport.
- Relojes vectoriales: Algoritmo de vector.

Algoritmos de exclusión mutua

- Solución centralizada
- Solución en anillo
- Solución descentralizada

Algoritmos de consenso

- Algoritmo de una ronda (One-round)
- Algoritmo de dos rondas (Two-round)
- Blockchain

Algoritmos de estado global del sistema

- Algoritmo de Chandy-Lamport

Algoritmos de elección de líder

- Algoritmo de selección en topología en anillo
- Algoritmo acosador (bully algorithm)

Sincronización de reloj

Algoritmo de Christian - Reloj físico

Un sistema distribuido con `i` nodos, cada uno tiene un reloj local `Ci`. Queremos que todos los relojes tengan la misma hora `t` y que sea la hora real.
Sin embargo, los chips de los relojes no son exactos. Tienen un pequeño error, por lo que tienden a ser cada vez más diferentes a `t`.
La solución es **sincronizar los relojes periódicamente de una fuente fiable**.

El algoritmo de Christian sirve para sincronizar un reloj local con el de un servidor fiable. Algunas consideraciones a tener en cuenta:

- El servidor proporciona una hora muy fiable, probablemente sincronizada con más servidores fiables.
- Los relojes **no** pueden volver atrás.
- La sincronización requiere paso de mensajes, pero consume tiempo.

Funcionamiento del algoritmo de Christian

1. El cliente pregunta la hora al servidor en t0 (cliente).
2. El servidor responde con el valor de su reloj en ese momento (tc).
3. El cliente recibe la hora en t1 (según su reloj).
4. C = tc + (t1-t0)/2

Si C es mayor que la hora actual del Cliente, entonces el Cliente tomará C como hora actual.
Si es menor, entonces el reloj del cliente se pausará durante (Cc-C) unidades de tiempo.

Análisis del algoritmo de Christian

El algoritmo de Christian asume que el **tiempo de transmisión es el mismo** para la petición y la respuesta, por lo que tc se obtiene justo en la mitad del proceso de comunicación.

Si uno de los mensajes dura más tiempo que el otro, la configuración no será correcta.
La **sincronización perfecta es imposible**, por lo que los programas tendrán que tolerar este error inherente al problema.

Algoritmo de Lamport - Reloj lógico

Los relojes lógicos son herramientas que sirven para establecer orden en eventos que ocurren, de tal forma que sean independientes de los relojes físicos. Son bastante útiles para saber **si un evento ha ocurrido antes que otro**. Tienen **sincronización perfecta, pero con ciertos límites**.

a -> b

a ocurre antes que b, y todos los nodos están de acuerdo en este orden.

Si a -> b & b -> c entonces a -> c

Aquí tenemos orden parcial, ya que puede haber concurrencia (x || y).

Funcionamiento del algoritmo de Lamport

1. Cada nodo tiene un contador que se incializa a 0.
2. Con cada envío de mensaje, el contador se incrementa.
3. Cada mensaje m se etiqueta (Cm) con el contador del emisor.
4. Cuando un nodo p recibe un mensaje, actualiza su reloj (Cp). **Cp = max (Cp, Cm) + 1**.
5. Al finalizar, tenemos un orden parcial. Este se puede convertir en total añadiendo el **id** del nodo como sufijo.

Sin embargo, en el caso de **a -> b** y **C(a) < C(b)**, no podemos asegurar **a -> b** o que **a || b**.

Relojes vectoriales

Para saber si **a->b** o **a || b**, dados **N** nodos, cada nodo **p** mantiene un Vp con N marcas de tiempo (timestamps). **V0[0] = k** es el número de eventos para 0.

1. Se asignan vectores iniciales para cada nodo [0,0,0].
2. Cuando se envía un mensaje, entonces el propio valor se actualiza y el vector se envía con el mensaje.
3. Cuando un nodo recibe un mensaje, incrementa su propio valor. Comprueba si los otros valores actuales son superiores que el valor actual y actualiza **Vp[i] = max(Vp[i], Vm[i])**.

- Un reloj es menor que otro si **V(a) < V(b)**
- Todos los componentes en V(a) son menores o iguales que los componentes en V(b), PERO uno de ellos es estrictamente inferior: **a->b**.
- Si **a -> b => V(a) < V(b)** y **V(a) < V(b) => a -> b**:

Si **V(a) < V(b) == false & V(b) < V(a) == false ==> a || b**

Exclusión mutua

Solución centralizada

Hay un líder que controla el acceso a la sección crítica. Concede permiso si está libre, y bloquea y anota los procesos que solicitan acceso cuando está ocupado.

Solución en anillo

No hay nodo que haga de líder, sino que hay un token que se van pasando entre ellos. Cuando un proceso quiera entrar a la sección crítica, debe esperar hasta tener el token para poder hacerlo. El nodo que tenga el token lo retendrá hasta que acabe la sección crítica.

Solución distribuida (descentralizada)

La exclusión mutua se resuelve con un mensaje broadcast. No hay nodo líder.
Cuando un nodo quiere entrar en la SC, avisa a los demás de que quiere acceder a ella. Si ningún otro nodo está en ella, le responden ACK y entra en la SC.
Si por el contrario, ya hay un proceso en la SC, este no responderá a la pregunta broadcast, por lo que el nodo que quiera entrar ahora esperará a que el otro proceso responda.
Cuando varios nodos quieran entrar a la vez en la sección crítica, hay que usar el algoritmo de Lamport para etiquetar cada intento y así controlar el orden.

Estado global del sistema: Chandy-Lamport

El estado global está formado por:

- El estado específico de cada nodo, con sus variables y valores.
- Mensajes enviados y pendientes.

Algunas de sus aplicaciones es recolectar basura (garbage collector) al ver objetos remotos que no se estén utilizando.

El escenario ideal sería capturar un momento en el tiempo (instantánea o snapshot) y preguntar a todos los nodos su información, pero no podemos lograr una sincronización perfecta. Por ello, usaremos una instantánea consistente (consistent snapshot).
Para esto necesitamos varios nodos, topología completa y canales fiables (FIFO o unidireccionales).

//Ver vídeos o diapositivas para ilustrarlo mejor

Elección de líder

Los líderes son necesarios en un sistema distribuido, porque simplifican los algoritmos y disminuyen el número de mensajes para acuerdos.
Tenemos que elegir un nuevo líder al inicio de la ejecución o cuando el líder actual no responde.

Bully's algorithm - Algoritmo abusón

0. Un nodo se da cuenta de que el líder no responde y notifica de que hay que elecciones con un **mensaje de elección**. Solo se comunica con nodos que tengan un identificador superior.
1. Los nodos activos responderán con ACK. Cuando un nodo recibe al menos un ACK, el nodo se retira del proceso de selección.
2. Se repite el paso anterior (enviar mensaje, esperar ACK) hasta que algún nodo no recibe un ACK y se convierte en el nuevo líder.

3. El nuevo líder se comunica con los demás con un **mensaje de coordinación**.

Topología en anillo

0. Un nodo se da cuenta de que el líder no responde y empieza el algoritmo. Este nodo será el inicializador y enviará un mensaje: `(ID inicializador, ID líder temporal)`.
1. Cuando un nodo recibe el mensaje, comprueba el ID del líder temporal y, si es menor que él, lo actualiza. Luego continúa enviando el mensaje al siguiente.
2. Se repite el paso anterior hasta que el mensaje llega al nodo inicializador, quien comunica al resto de nodos quién es el nuevo líder con un **mensaje de coordinación**.

Consenso: Blockchain

El problema del **acuerdo bizantino** requiere de un proceso líder, con un valor inicial, que debe llegar a un acuerdo con otros procesos sobre su valor inicial. En ausencia de fallo no tiene interés. Dos tipos de fallo:

- **Crash failure**. Un nodo deja de funcionar, lo que puede causar que el resto de nodos no lleguen a un consenso.
- **Byzantine failure**. Un nodo tiene un comportamiento anómalo, incluso malintencionado.

Sujeto a las siguientes condiciones:

- **Acuerdo**. Todos los procesos que no fallan deben estar de acuerdo en el mismo valor.
- **Validez**. Si el proceso líder no falla, entonces el valor acordado por el resto de procesos debe ser el inicial del líder.
- **Terminación**. Todo proceso que no falle debe tomar una decisión.

El problema del consenso

Difiere del acuerdo bizantino en que cada proceso tiene un valor inicial y todos los procesos que no fallan deben estar de acuerdo en un valor de esos. Debe satisfacer:

- **Acuerdo**. La decisión final de todo proceso que no falla debe ser la misma.
- **Validez**. Si todo proceso que no falla empieza con el mismo valor v, entonces la decisión final debe ser v.
- **Terminación**. Todo proceso que no falle debe tomar una decisión.

El problema de la consistencia interactiva

Difiere del acuerdo bizantino en que cada proceso tiene un valor inicial y todos los procesos correctos deben acordar un conjunto de valores, con un valor para cada proceso. Se debe cumplir:

- **Acuerdo**. Todos los procesos que no fallan deben llegar a un acuerdo en los mismos valores para el array A[v1...vn].
- **Validez**. Si el proceso i no falla y su valor inicial es Vi, entonces todos los procesos que no fallan deben estar de acuerdo en Vi como el i-ésimo elemento del array A. Si el proceso j falla, entonces los procesos que no fallan pueden acordar cualquier valor para A[j].
- **Terminación**. Cada proceso que no falla debe eventualmente tomar una decisión sobre el array A.

Tema 8. Plataformas de componentes distribuidos

- Los sistemas distribuidos de amplia escala están compuestas actualmente por microservicios. Permiten bajo acoplamiento e incluso desarrollo en múltiples lenguajes de programación. Permiten escalabilidad (cores, dispositivos, nodos, clusters, data-centers).
- Comunicación fundamentalmente estructurada como RPC.
    - Múltiples nodos de comunicación RPC.
    - Terminología habitual: cleintes usan los stub para llamar a los métodos en los servicios/servidores.
    - Facilidad para usar interfaces (síncronos y/o asíncronos) a través del código generado de soporte.
- Bastante documentación relacionada con la descripción de sistemas RPC.
- **Local Procedure Call** ejecuta localmente.
- **Remote Procedure Call** ejecuta remotamente.

gRPC

Google ya ha tenido cuatro generaciones de sistemas RPC internos: los Stubby.

- Todas las aplicaciones de producción y sistemas se producen usando RPC.
- Alrededor de 10^{10} invocaciones RPC por segundo.
- API escritas en distintos lenguajes tales como C++, Java, Python o Go.

A través de gRPC introduce las prácticas aprendidas internamente relacionadas con escalabilidad, rendimiento o desarrollo de API.

- Definición sencilla de servicios: Protocol buffers y generación automática del código de soporte.
- Gran escalabilidad: Protocol buffers (again) y HTTP/2.
- Soporte de múltiples lenguajes de programación.
- Utilizado por grandes empresas.

Protocol Buffers

- Protocol Buffers es el lenguaje neutral para *serializar* datos: RPC y almacenamiento.
- Codifica los datos en **formato binario**. Es mucho más rápido y más ligero en comparación a JSON.
- Ventajas obtenidas por HTTP/2.
- Generadores de código para múltiples lenguajes.
- Fuertemente tipado.

Conceptos gRPC

Tipos de comunicación

- **Unary RPC**. El cliente realiza una petición RPC y obtiene una respuesta.
- **Server streaming RPC**. El cliente envía una petición y el servidor responde con un flujo (stream) de mensajes. El cliente leerá hasta que no queden más mensajes en el flujo.
- **Client streaming RPC**. El cliente envía un flujo (stream) de mensajes al servidor. El servidor leerá todos los datos de la secuencia de mensajes y responderá con un tipo de dato de retorno. El cliente

leerá hasta que no queden más mensajes en el flujo.
- **Bidirectional streaming RPC**. Ambos lados de la comunicación intercambian secuencias de mensajes. Los flujos operan de forma independiente, de forma que los clientes y servidores pueden leer y escribir en cualquier orden. **Se garantiza el orden** de los mensajes en cada flujo.

Ciclo de vida RPC

- **Deadline/Timeouts**. gRPC permite a los clientes especificar cuánto tiempo están dispuestos a esperar para que una RPC se complete antes de que la RPC se termine con un error DEADLINE_EXCEEDED.
- **RPC Termination**. Los clientes y servidores especifican de forma independiente en qué momento ha terminado la invocación.
- **Canceling RPC**. Tanto clientes como servidores pueden cancelar las llamadas RPC en cualquier momento.

Información de comunicación

- **Metadata**. Los metadatos son información sobre una llamada RPC determinada en forma de una lista de pares clave-valor.
- **Channels**. Un canal gRPC proporciona una conexión gRPC en un host y puerto especificados. Se utiliza al crear un código auxiliar de cliente. Los clientes pueden especificar el canal argumento para modificar el comportamiento predeterminado de gRPC, como cambiar mensaje Comprensión activada o desactivada.