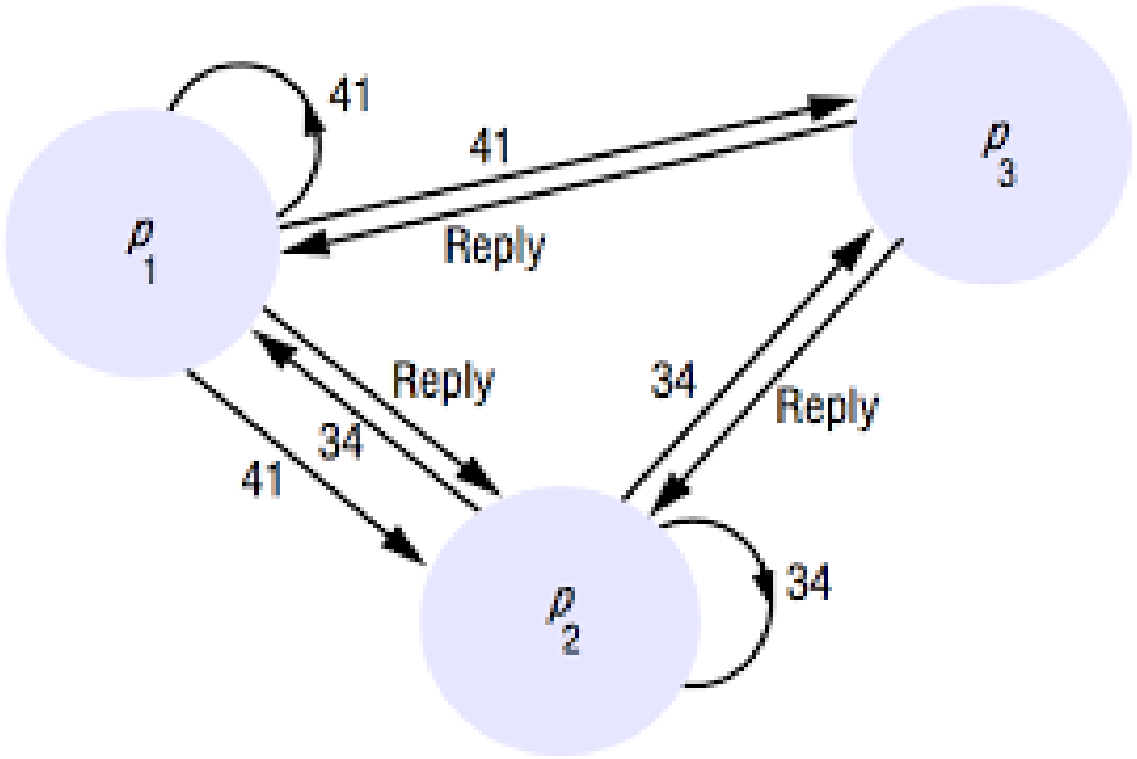
Informe de Desarrollo de la Práctica Obligatoria de Sistemas Distribuidos

Algoritmo de Ricart y Agrawala



Autores

Raúl Melgosa Salvador
Ángel Torijano Sexmero

Índice

Introducción

La práctica a desarrollar tiene como objetivo implementar un conjunto de procesos distribuidos que autorregulen el acceso a una zona de exclusión mutua común. Dicho de otra manera, el objetivo fundamental de esta práctica es implementar una sección crítica distribuida a la que accedan distintos procesos desde distintos equipos.

Para la implementación de la sección crítica, se va a utilizar el algoritmo de Ricart y Agrawala, el cuál se apoyará en los tiempos lógicos de Lamport para su funcionamiento.

En la práctica, se van a disponer de 6 procesos ejecutados en 3 máquinas distintas los cuales van a ejecutar un mismo código. Cada proceso tendrá que realizar los siguientes pasos:

- Simular un cálculo con una duración entre 0.3 y 0.5 segundos
- Entrar en la sección crítica distribuida
- Simular un cálculo con una duración entre 0.1 y 0.3 segundos
- Repetir los 3 pasos anteriores 100 veces
- Terminar la ejecución de manera ordenada

Cada vez que un proceso entre o salga de la sección crítica, escribirá un mensaje en un log indicando el número de proceso que accede a la sección crítica, el tipo de operación (entrada o salida) y el instante en el que se ha producido la entrada o salida (en milisegundos).

Cada proceso escribirá esto en un fichero de log distinto. Al finalizar la ejecución, se deberán fusionar todos los ficheros de log en un único fichero y se deberá comprobar que no se ha violado la sección crítica.

Como los relojes de las distintas máquinas donde se ejecutan los procesos no están sincronizados, se deberá calcular los desvíos temporales con respecto a una máquina de referencia. Para ello, se utilizará el algoritmo NTP de forma que un proceso que se ejecute desde la máquina de referencia obtendrá la desviación temporal con respecto a las otras máquinas. Ese desvío temporal se tendrá en cuenta a la hroa de fusionar los log y verificar si ha habido o no violaciones en la sección crítica.

En la siguiente imagen se presenta el esquema fundamental del planteamiento a realizar en la práctica:

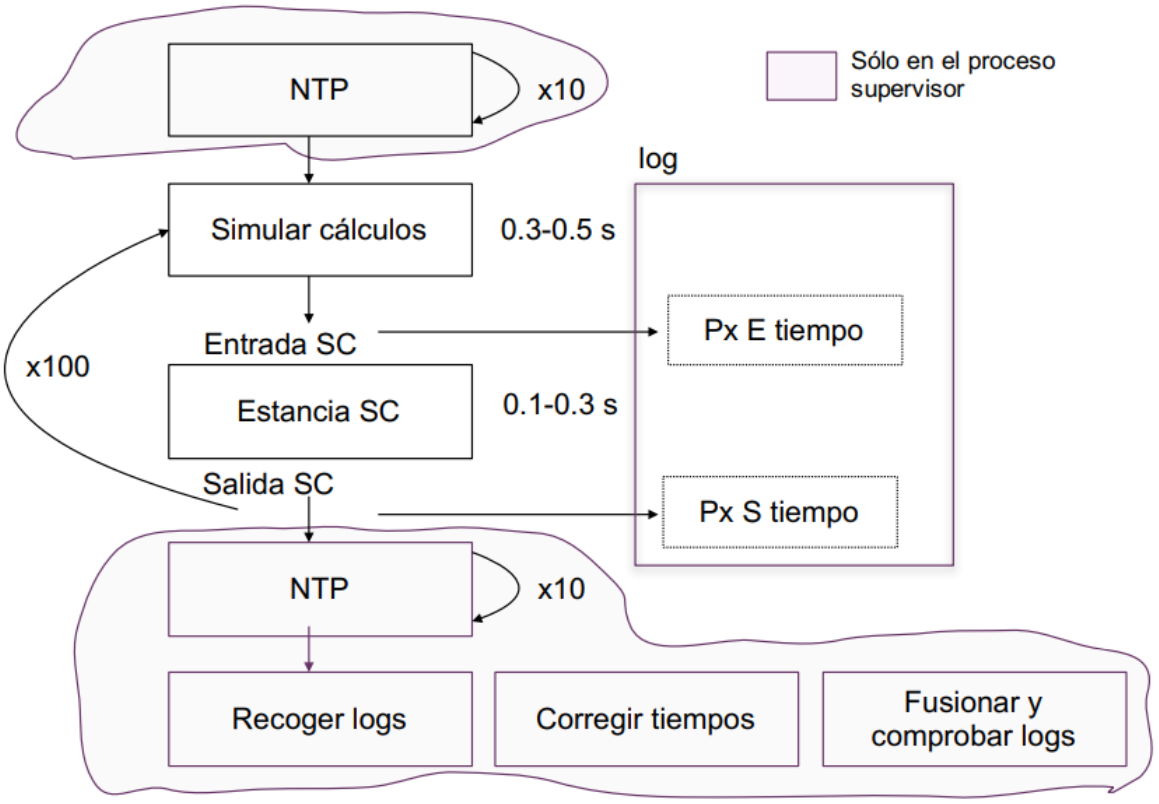


Figura 1: Esquema fundamental de la práctica

Estructura

En la práctica, se van a contar con 6 procesos los cuales se van a ejecutar en 3 máquinas distintas por lo que cada máquina ejecutará 2 procesos. Todos los procesos deben poder ejecutarse desde un único nodo.

También hay que tener en cuenta que cada proceso tendrá aspectos tanto como de servidor como de cliente ya que enviará mensajes a otros procesos y recibirá mensajes de otros procesos.

Para cumplir con los requerimientos de la práctica, se ha planteado la siguiente estructura:

- Los procesos contarán una serie de métodos REST los cuales podrán ser accedidos por el resto de procesos (carácter de servicio).
- Un método run desde el cual se realizan distintas peticiones REST de forma que podrán crearse múltiples instancias o hilos (carácter de cliente).

Con esto se consigue otorgar a los procesos un carácter de servicio y un carácter de servidor.

Por una parte, en cada máquina se desplegará el proceso en 2 servidores Tomcat de manera que cada servidor Tomcat escuchara por un número de puerto diferente (del 8080 al 8085). Por otra parte, desde una máquina central o de referencia se ejecutarán 6 hilos del

proceso, los cuales se conectarán con los 6 procesos servidores a través de las peticiones REST.

A través de este planteamiento, se tienen 6 procesos ejecutándose en 3 máquinas distintas. Además, todos los procesos pueden ser ejecutados desde una misma máquina siempre y cuando los procesos servidores estén correctamente desplegados en los servidores Tomcat de las distintas máquinas.

En la siguiente imagen se adjunta un pequeño esquema de la estructura de la práctica realizada:

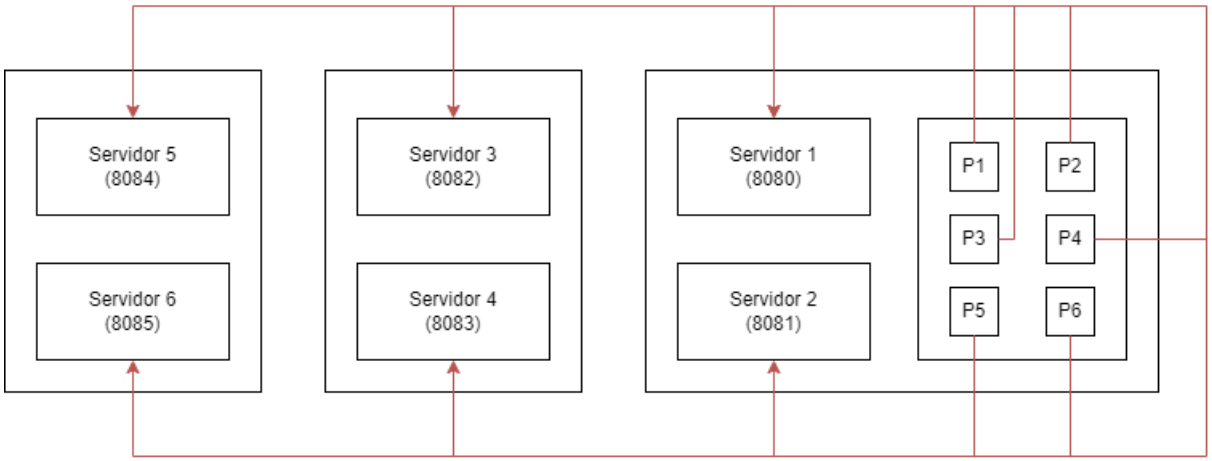


Figura 2: Esquema de la estructura fundamental de la práctica

Clases

La práctica se implementa en un Dynamic Web Project de Java, el cual se creará desde Eclipse. Este proyecto utilizará las librerías de Jersey de la versión 2.39 y para los servidores se utilizaran servidores Tomcat de la versión 8.5.

Este proyecto contará con 5 clases fundamentales:

- **Servidor.java**: Es la clase fundamental la cual incluye las funciones REST (carácter de servicio) como el método run y métodos auxiliares los cuales son llamados dentro de este método run (carácter de cliente).
- **ServicioArranque.java**: Esta clase se encarga de crear los 6 hilos los cuales se conectaran con los servidores a través de las peticiones REST.
- **Par.java**: Clase auxiliar la cual permite agrupar y almacenar los valores del offset y delay obtenidos en el algoritmo NTP.
- **Peticion.java**: Clase auxiliar la cual permite agrupar y almacenar los valores del tiempo e identificador de proceso, los cuales conforman una petición en el algoritmo de Ricart y Agrawala.
- **Comprobador.java**: Clase auxiliar la cual recorre el log fusionado y corregido en busca de posibles violaciones en la sección crítica.

Funcionamiento

En primer lugar, se crearán los 6 procesos clientes a través de ServicioArranque.java. En la creación de cada proceso cliente se recibe como parámetro el identificador del proceso de manera que cada proceso contará con un identificador distinto (en nuestro caso los identificadores van del 0 al 5). También se recibe como parámetros las direcciones IP de las 3 máquinas.

Se inicializan las URIs de cada proceso cliente (según las IDs que se le han pasado se abriran en su determinado puerto con su determinada IP) y se limpian los logs asociados a cada proceso mediante una petición REST. Estos logs estarán situados localmente en la máquina en la que se haya desplegado cada proceso servidor.

Explicaremos más adelante el despliegue en profundidad.

```
for (int i = 0; i < numero_procesos; i++) {
    if (i < 2) {
        client[i] = ClientBuilder.newClient();
        uri[i] = UriBuilder.fromUri("http://" + IP1 + ":808" + i + "/Obligatoria").build();
        target[i] = client[i].target(uri[i]);
    }
    else if (i >= 2 && i < 4) {
        client[i] = ClientBuilder.newClient();
        uri[i] = UriBuilder.fromUri("http://" + IP2 + ":808" + i + "/Obligatoria").build();
        target[i] = client[i].target(uri[i]);
    }
    else {
        client[i] = ClientBuilder.newClient();
        uri[i] = UriBuilder.fromUri("http://" + IP3 + ":808" + i + "/Obligatoria").build();
        target[i] = client[i].target(uri[i]);
    }
}
```

Figura 3: Inicialización de las URIs

A continuación se inicializan las URIs de los procesos servidores mediante otra petición REST. Esto se realiza ya que los procesos servidores cuentan con variables independientes a las de los procesos clientes. Una vez hecho esto controlaremos que todos los procesos hayan sido creados con un semáforo parecido al que usamos para el “Preparados” del ejercicio de los 100 metros lisos.

```java
@GET // tipo de petición HTTP
@Produces(MediaType.TEXT_PLAIN)
@Path("preparado")
public void preparado()
{
    try {
        semaforo_proteccion.acquire(1);
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    num_preparados = num_preparados + 1;
    if (num_preparados < numero_procesos) {
        try {
            semaforo_proteccion.release(1);
            semaforo_preparados.acquire(1);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
    else {
        num_preparados = 0;
        semaforo_proteccion.release(1);
        semaforo_preparados.release(numero_procesos - 1);
    }
}
```

Figura 4: Espera a la creación de todos los procesos

Ahora que todos están preparados, el proceso con ID = 0 se encargará de aplicar el algoritmo NTP, el cual calculará el offset y el delay respecto a los otras 2 máquinas ~líneas 133-177~. Después volvemos a sincronizar con otro semáforo bajo la petición REST “Listo”, con esta petición REST “Listo” se busca que los demás procesos esperen a que el proceso 0 realice el algoritmo NTP .

Una vez sincronizados, empezamos a hacer el algoritmo de Ricart y Agrawala. Al ser peticiones dentro de una petición, vamos a necesitar iniciar una URI por cada servidor ~líneas 555-575~. Iniciamos la sección crítica y vaciamos la cola. Simulamos los cálculos y avisamos de que queremos entrar en la sección crítica. Posteriormente hacemos multidifusión de las peticiones de forma asíncrona (Num_procesos - 1 veces) y las cargamos en la cola (en el caso de que se den las condiciones) en una lista de la clase Peticion ~Línea 602~.

Para realizar la multidifusión de peticiones de entrada a la sección crítica, cada proceso va a llamar a la petición REST “enviarPeticion” de todos los procesos servidores excepto al servidor asociado al cliente correspondiente (no se va a enviar una petición a uno mismo).

```java
@GET
@Produces(MediaType.TEXT_PLAIN)
@Path("enviarPeticion")
public void enviarPeticion(@QueryParam("Tj") long Tj, @QueryParam("Pj") long Pj)
{
    try {
        semaforo_antibloqueos.acquire(1);
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    if (Ci >= Tj) {
        Ci = Ci + 1;
    }
    else {
        Ci = Tj + 1;
    }
    if (estado == "TOMADA" || (estado == "BUSCADA" && (Ti < Tj || (Ti == Tj && Pi < Pj)))) {
        Peticion p = new Peticion(Tj, Pj);
        cola.add(p);
    }
    else {
        //Responde inmediatamente a Pj
        target[(int) Pj].path("rest").path("hola").path("enviarRespuesta").request(MediaType.TEXT_PLAIN).async().get(String.class);
    }
    semaforo_antibloqueos.release(1);
}
```

Figura 5: Enviar petición dentro del Algoritmo de Ricart y Agrawala

Cada petición estará controlada por un semáforo ya que estas variables pueden ser accedidas por varios procesos al mismo tiempo y producir interbloqueos. De esta manera, el semáforo busca proteger los accesos a las variables compartidas y procesar las peticiones de manera ordenada.

Para responder a una petición, se utilizará la función REST “enviarRespuesta” la cual incrementará el valor de un semáforo de respuestas en una unidad.

Después de la multidifusión de las peticiones, cada proceso esperará a que el semáforo de respuestas llegue al número de procesos menos uno, lo que significa que el resto de procesos habrá respondido a la petición.

```java
@GET
@Produces(MediaType.TEXT_PLAIN)
@Path("esperarRespuestas")
public void esperarRespuestas()
{
    try {
        semaforo_respuestas.acquire(numero_procesos - 1);
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
```

Figura 6: Esperar respuestas dentro del Algoritmo de Ricart y Agrawala

Hecho esto ya, entra en la sección crítica el proceso que le corresponda. Se realizan las escrituras en cada log correspondiente y se simulan los cálculos necesarios. Finalmente salimos de la sección crítica y se limpia la cola de peticiones enviando tantas respuestas como sea el tamaño que tenga la cola. Todo este proceso se repite el número de interacciones que hayamos decidido, en este caso 100 ~Líneas 559-664~

Después del algoritmo de Ricart y Agrawala volvemos a hacer NTP con el proceso 0. Una vez acabado calculamos cuales son los mejores pares de offset y delay. Tenemos la manera rudimentaria y Marzullo implementados, siendo el segundo método el que prevalece (en caso de que se quiera usar el primero únicamente bastará con comentar la línea donde se invoca Marzullo).

```
//calculamos la media del mejor par inicial y el mejor par final para cada proceso
for (int m = 0; m < (numero_procesos / 2) - 1; m++) {
    mejorPar[m] = new Par();

    mejorPar[m].setO((mejorParInicial[m].getO() + mejorParFinal[m].getO()) / 2);
    mejorPar[m].setD((mejorParInicial[m].getD() + mejorParFinal[m].getD()) / 2);

    mejorPar[m] = marzullo(pares[m]); //realmente se va a aplicar Marzullo
                  //para encontrar el mejor par pero se deja implementada la otra solucion.
                  //Para utilizar la otra solucion simplemente hay que comentar esta linea

    System.out.println("El valor del mejor par medio de la maquina "
                + m + " es de (" + mejorPar[m].getO() + ", " + mejorPar[m].getD() + ") [formato (offset, delay)]");
}
```

Figura 7: Obtención del mejor par para cada máquina

Finalmente necesitamos recibir todos los ficheros log y que el proceso 0 se encargue de juntarlos en un único log ~Líneas 257-345~. Para eso, cada proceso leerá localmente su fichero log asociado y guardará el contenido de cada log en una variable String[] con la función REST “leerFichero” ~~Líneas 748-782~. Cada elemento del vector de Strings será un fichero completo escrito línea a línea separadas por “\r\n” que corresponderá al ID de cada uno. Después, todos los procesos enviaran al proceso 0 el contenido de los log correspondientes a través de la función REST “recibirLog” y el 0 creará un copia local de cada uno de los ficheros log de los procesos.

Una vez hecho esto, esperaremos a que todos los procesos acaben de enviar el contenido de los log y en el caso del proceso 0 recibir los log y realizar NTP con un último semáforo de sincronización (semáforo fin).

Será entonces el turno del proceso 0, que le tocará guardar en su máquina todos los logs y consolidarlos en uno solo. Primero se limpia el log.txt con la función REST “limpiarLog”. Después vamos leyendo uno a uno los logs y almacenando su contenido en un ArrayList de Strings. Para calcular los tiempos reales acordes con cada máquina, primero diferenciamos de qué máquina viene comparando el inicio de cada línea. Para la máquina 1 empieza por P3 o P4 y para la máquina 2 empieza por P5 o P6. Ahora separamos por tokens la línea y ya podemos calcular el tiempo restando el offset del mejor par de cada máquina. Este tiempo junto con el resto de la línea del log original se almacenará en la lista de Strings. Ahora con los tiempos reales corregidos con respecto a la máquina de referencia podemos ordenar las líneas según los tiempos reales corregidos. Esto lo hacemos con un Collections.sort al ArrayList de los logs.

Tenemos totalmente almacenados todos los logs en nuestra ArrayList de Strings ordenados y ahora podemos comprobar si han habido fallos en la sección crítica. Para ello, primero reescribimos el fichero de log general con el contenido del Arraylist que contiene la corrección de tiempos de las entradas y salidas a la SC correspondientes y la ordenación de dichas entradas y salidas a la SC en función de los tiempos corregidos.

Finalmente, el proceso 0 llamará a la clase “Comprobador” la cual se encargará de detectar si se han producido o no violaciones en la sección crítica distribuida. Este comprobador recibe 3 parámetros o argumentos los cuales son la ruta del log.txt y el valor del delay del mejor par calculado a través de NTP de las otras dos máquinas.

Despliegue

Para realizar el despliegue de la práctica entregada se deberá hacer lo siguiente:

- En las 3 maquinas crear un Dynamic Web Project que incluya las librerías de Jersey necesarias e incluya los 5 ficheros java antes mencionados.
- En la máquina de referencia o principal, crear 2 servidores Tomcat que utilicen los puertos 8080 y 8081 respectivamente.
- En la segunda máquina, crear 2 servidores Tomcat que utilicen los puertos 8082 y 8083 respectivamente.
- En la tercera máquina, crear 2 servidores Tomcat que utilicen los puertos 8083 y 8084 respectivamente.
- Ejecutar la clase Servidor.java en cada uno de los servidores creados en cada una de las máquinas.
- Ejecutar ServicioArranque.java como Aplicación de Java desde la máquina de referencia, pasando como parámetros las direcciones IP de las 3 máquinas involucradas

Tras la ejecución de la práctica, se almacenará localmente en la máquina de referencia los ficheros log generados por los 6 procesos así como el log general fusionado y corregido. Estos log se encontrarán en la ruta personal del equipo. Por otra parte, las otras 2 máquinas almacenarán localmente los ficheros log correspondientes a los 2 procesos servidores que se están ejecutando en dichas máquinas. Estos log también se encontrarán en la ruta personal de la máquina correspondiente.

Bibliografía

Para la realización de la práctica, se han utilizado como guía las siguientes páginas y artículos:

Para comprender y realizar la implementación del algoritmo de Marzullo:

- https://es.wikipedia.org/wiki/Algoritmo_de_Marzullo
- https://www.atc.uniovi.es/inf_superior/4atc/DISTRIBUIDAS/05-L11-Sincronizacion.pdf

Para escribir y leer de ficheros en Java:

- https://chuidiang.org/index.php?title=Lectura_y_Escritura_de_Ficheros_en_Java#:~:text=Podemos%20abrir%20un%20fichero%20de,y%20extraer%20campos%20de%20ella.
- https://es.stackoverflow.com/questions/138958/c%C3%B3mo-escribir-en-un-archivo-de-texto-en-java