**Práctica obligatoria**

**Sistemas Distribuidos 19-20**

**Algoritmo de *Ricart y Argawala***

Francisco Pinto Santos

Héctor Sánchez San Blas

Índice

[1. Introducción 3](#_Toc38136469)

[2. Aspectos relevantes 4](#_Toc38136470)

[2.1. Políticas y formato de logging 4](#_Toc38136471)

[2.2. Política de nombrado de hilos 4](#_Toc38136472)

[2.3. Configuración del sistema 5](#_Toc38136473)

[2.4. Patrones utilizados en la implementación de servicios 5](#_Toc38136474)

[2.5. Localización de los procesos por id dentro del sistema 6](#_Toc38136475)

[3. Sincronización 7](#_Toc38136476)

[3.1. Inicio y finalización del sistema 7](#_Toc38136477)

[3.2. Intento de acceso a sección critica 8](#_Toc38136478)

[3.3. Acceso a la sección critica 9](#_Toc38136479)

[3.4. Gestión de “estado de proceso” en el servicio de sección critica 10](#_Toc38136480)

[3.5. “Encolado” de procesos en la sección critica 10](#_Toc38136481)

[3.6. Problemas en el desarrollo 12](#_Toc38136482)

[4. Aspectos relevantes de NTP 14](#_Toc38136483)

[5. Explicación del contenido de los paquetes 15](#_Toc38136484)

[5.1. Paquete com.ssdd.cs.bean 15](#_Toc38136485)

[5.1.1. CritricalSectionState.java 15](#_Toc38136486)

[5.1.2. LamportCounter.java 15](#_Toc38136487)

[5.2. Paquete com.ssdd.cs.client 15](#_Toc38136488)

[5.2.1. CritricalSectionClient.java 16](#_Toc38136489)

[5.2.2. CritricalSectionRouter.java 16](#_Toc38136490)

[5.3. Paquete com.ssdd.cs.main 16](#_Toc38136491)

[5.3.1. Main.java 16](#_Toc38136492)

[5.3.2. MainLogVerification.java 16](#_Toc38136493)

[5.3.3. MainSimulation.java 16](#_Toc38136494)

[5.3.4. MainSupervisor.java 17](#_Toc38136495)

[5.4. Paquete com.ssdd.ntp.client 17](#_Toc38136496)

[5.5. Paquete com.ssdd.simulation 17](#_Toc38136497)

[5.5.1. SimulationProcess.java 17](#_Toc38136498)

[5.5.2. SimulationProcessBuilder 17](#_Toc38136499)

[5.5.3. SimulationLog.java 17](#_Toc38136500)

[5.5.4. SimulationLogAdjuster.java 17](#_Toc38136501)

[6. Despliegue 18](#_Toc38136502)

# Introducción

Este informe explicara la sincronización y los esquemas de comunicación definidos, ya que si se desea conocer algún especto mas especifico, este puede ser consultado en el *javadoc* del proyecto ubicado en la carpeta *doc* de este.

# Aspectos relevantes

## Políticas y formato de logging

Para el logging, se ha utilizado la clase *java.util.Logger* y se ha seguido la buena práctica de establecer uno por clase, construido en la clase *com.ssdd.util.logging.SSDDLogFactory*.

En cuanto al formato de este, se ha definido mediante un *java.util.formatter* definido en la clase *com.ssdd.util.logging.SSDDLogFactory* con el siguiente formato:

*[fecha y hora] [nivel de logging] [nombre del hilo] [clase] mensaje de logging*

Para manejar la persistencia del log, en la clase *com.ssdd.util.logging.SSDDLogFactory* se ha desarrollado la construcción de un *java.util.logging.Handler*. Este dependiendo de la configuración podrá tratarse de un *java.util.logging.ConsoleHandler*, o bien un *com.ssdd.util.logging.centralized.client.CentralizedLogHandler*.

Este último, se trata de un *Handler* que enviara todo el logging a el servicio de *logging* centralizado, para ser persistido a un fichero.

En cuanto a el nivel de logging, se ha establecido a INFO, en todos los casos en los que la información no es relevante, y a WARNING, en el caso de los errores.

Por último, cabe destacar el aspecto mas

Todos los aspectos descritos en este punto pueden ser configurados mediante la clase de constantes *com.ssdd.util.constants.ILoggingConstants*.

Por último, del proceso de logging centralizado, cabe destacar que se comprueba el estado del servicio de *logging* centralizado antes de proceder, y en el caso de que este falle, se procede a un *logging* local.

## Política de nombrado de hilos

Se ha procurado que todos los hilos de la aplicación tengan nombre, para facilitar las tareas de depurado y visualizar como se desarrolla tanto la simulación como el NTP en el log. Para lo cual, se han establecido los siguientes formatos:

* NTP:
  + cliente: *main*.
  + servidor: *NTP*.
* Sección critica:
  + Cliente: Pi (con i, el id de uno de los procesos del sistema).
  + Servidor: CS en el caso de tratarse de un endpoint en el que el proceso no es relevante y CS.i (con i, el id de uno de los procesos del sistema) en el caso contrario.
* *Thread Pools*: como se han usado *thread* pools, se ha definido el siguiente formato: <nombre hilo padre>.<id de *worker*>, siendo el id del *worker* un numero que va de 0 a N-1 siendo N el número de *workers*.

## Configuración del sistema

Todas las configuraciones para el sistema se pueden encontrar en el paquete *com.ssdd.util.constants*, repartidas en las siguientes clases:

* *IConstants*: Constantes generales del sistema que no encajan en ninguna otra clase de definición de constantes.
* *INtpConstants*: constantes relativas a el proceso de muestreo y calculo de los desfases entre maquinas por NTP.
* *ILoggingConstants*: constantes relativas a los logs como pueden ser nivel de logging, centralizado o no, ip del servicio que aloja el log centralizado, fichero de logging centralizado, etc.
* *ISimulationConstnats*: constantes relativas a la simulación, como numero de iteraciones, tiempo de calculo dentro y fuera de la sección critica, etc.

## Patrones utilizados en la implementación de servicios

Para realizar la comunicación con los servicios, se ha implementado el patrón Proxy, para abstraer las llamadas a las API.

Además, para realizar las llamadas al proxy y el tratamiento de la información recibida de este, se ha utilizado una clase cliente del proxy para aportar un mayor grado de abstracción.

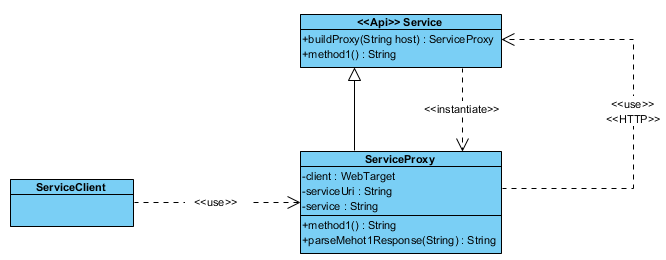


Figura 2.4.1

En cuanto a la implementación de servicios, cabe destacar también que se ha establecido un *endpoint* *status* en cada servicio, que devuelve un String con el estado del servicio para verificar que dicho servicio está activo cuando se desee comprobar por razones diversas como puede ser *debbuging*.

## Localización de los procesos por id dentro del sistema

Para evitar implementar de forma estática qué proceso se asocia a qué servicio, se han definido los siguientes mecanismos:

* Cada proceso al inicio del sistema realiza un proceso de asociación con un servicio, para indicarle a este, que será el responsable de “guardar su estado”.

El servicio al cual se suscribe se le indica por parámetro al iniciar la simulación.

* Cada servicio implementa un *endpoint* llamado *suscribed* a través del cual los procesos pueden solicitar la información sobre que procesos están suscritos a dicho servicio.
* Cuando un proceso desea establecer comunicaciones con otro, consulta una instancia de *CriticalSectionRouter*, que contiene un *Hasmap* de asociación entre id de proceso y *Proxy* a el servicio al que está suscrito ese proceso.

En el caso de encontrar el id del proceso en el mapa, devuelve un *Proxy* a la sección critica de ese proceso. En caso contrario, realiza una petición a cada servicio del sistema (al *endpiont suscribed*), para obtener que procesos están suscritos y actualizar así la información de su *Hashmap*, tras lo cual devuelve el *Proxy* al servicio solicitado para que el proceso realice las comunicaciones pertinentes.

Este sistema no puede fallar porque no encuentre un proceso no se haya suscrito, debido a que todos los procesos realizan el proceso de asociación a un servicio (suscripción) antes de llamar al *endpoint* *ready* (que los hace esperar hasta que todos han llamado a dicho endpoint), por lo que cuando todos comienzan a enviarse mensajes, todos los procesos ya han sido asociados a un servicio del sistema.

# Sincronización

## Inicio y finalización del sistema

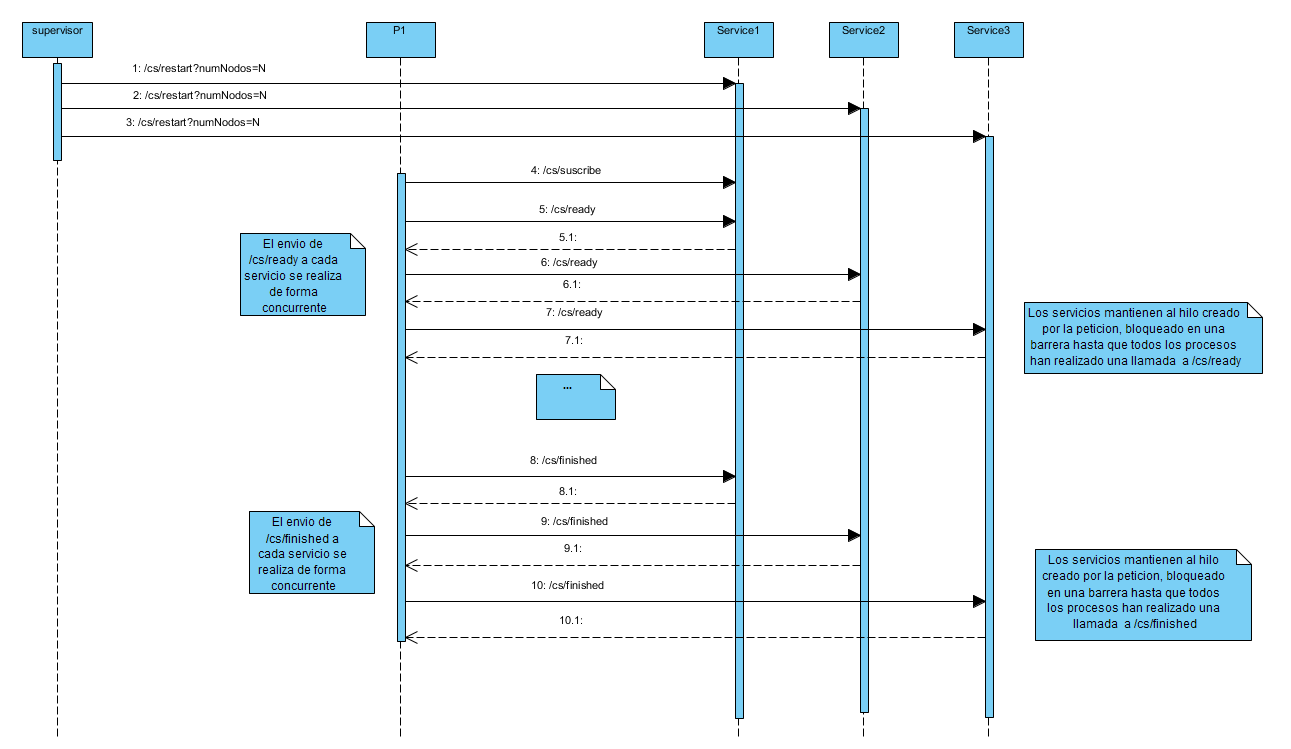


Figura 3.1.1

El esquema seguido para inicializar y finalizar el sistema se describe en la figura 3.1.1, no obstante, los pasos se describen a continuación:

* El supervisor reinicia los servicios de sección critica en cada servidor, indicándole el número de procesos que intentarán acceder a la sección critica.
* Cada proceso (en este caso solo representado P1 para evitar la congestión por mensajes en el diagrama y facilitar así su entendimiento), realiza un proceso de asociación a un servicio, el cual normalmente estará corriendo en la misma máquina que el proceso. Esto se hace para que el servicio registre que tiene un proceso asociado e instancie un “estado de proceso” dentro de su contexto.
* Tras ello, cada proceso realiza de forma concurrente una llamada al *endpoint* de *ready* de cada servicio del sistema y se queda bloqueado esperando que todos los servicios le respondan, cosa que ocurrirá cuando todos los procesos hayan realizado una llamada a dicho endpoint en cada servicio.

Para hacer esto del lado del servicio, se bloque al hilo que se crea con cada proceso en una barrera tipo *java.util.concurrent.CyclicBarrier*.

* Una vez completado esto, ya inician todos los procesos la simulación (cálculos, acceso a la sección critica, *logging*, …).
* Por último, cuando cada proceso finaliza, realiza de forma concurrente una llamada al *endpoint* de *finsihed* de cada servicio del sistema y se queda bloqueado esperando que todos los servicios le respondan, cosa que ocurrirá cuando todos los procesos hayan realizado una llamada a dicho endpoint en cada servicio.

Para hacer esto del lado del servicio, al igual que se ha realizado anteriormente en *ready*, se bloque al hilo que se crea con cada proceso en una barrera tipo *java.util.concurrent.CyclicBarrier*.

Estos dos procesos, pueden parecer redundantes, no obstante, tienen su justificación:

* Empezar todos a la vez: si no establecemos esta sincronización, de la manera que tenemos implementado el sistema, si un proceso desea enviar un mensaje a otro, consultara su *CriticalSectionRouter*, no obstante, este por debajo realizara llamadas a el *path /ssdd/cs/suscribed*, para conocer que procesos están asociados a cada servidor cuando se realice la primera llamada a este. En el caso de no encontrar un proceso, este devolverá *null*, lo cual hará que se interrumpa la ejecución porque no se ha encontrado un proceso.
* Terminar todos a la vez: en el script *deploy.bash* cuando se desea ejecutar una simulación, es necesario correr 3 instancias de simulación (una en cada máquina) con dos procesos cada una. Para poder realizar esto, se ejecuta de forma remota el comando “*java -jar ssdd.jar …*” en segundo plano (con el símbolo &) para dos de las máquinas y en primer plano para el restante.

De esta forma, si no se define un mecanismo de espera, la maquina en la que no se ha ejecutado un proceso en segundo plano, podría terminar antes, provocando que en el script se continúe con el cálculo de NTP, corrección de logs, etc. sin haber terminado los otros dos procesos restantes.

## Intento de acceso a sección critica

Para realizar un acceso a la sección critica, se utiliza el algoritmo de *Ricart* y *Argawala*, pero los *endpoints* establecidos para realizar este proceso en los servicios son los siguientes:

* */set/requested*: permite a un proceso establecer en su servicio asociado el estado de su sección critica a *REQUESTED*.

Además, tiene dos funciones más: La primera de ellas se trata de establecer a “permitido” el proceso de “encolar” (poner a esperar a los procesos con una marca de tiempo mayor a la del servicio actual).

La segunda función de este es obtener la marca de tiempo que llevara el mensaje de petición (representado como *Ti* en el enunciado) y almacenarlo en el estado del servidor.

* */request*: permite realizar una petición de acceso a la sección critica de un proceso a otro.

En el caso de que la peticion sea

* */set/acquired*: permite a un proceso establecer en su servicio asociado el estado de su sección critica a *ACQUIRED*, además, actualiza su contador de *Lamport* asociado, incrementándolo en una unidad.

Tiene una última función, que es liberar a todos los procesos “encolados” (puestos a esperar durante el proceso de petición) y establecer a “denegado” el proceso de “encolado”.

* */release*: establece la sección critica a liberada y permite el acceso a todos los procesos que estén esperando a acceder a esta.

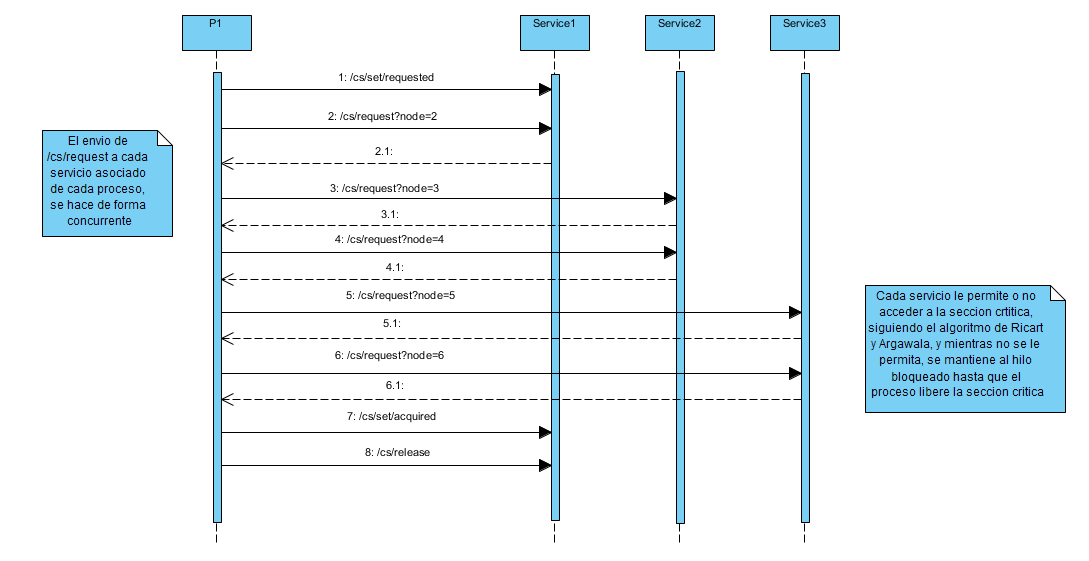


Figura 3.2.1

## Acceso a la sección critica

Cuando un proceso desea acceder a la sección critica, tiene que realizar una petición al endpoint */request* a cada uno del resto de procesos del sistema\*. Para ello, se utiliza un *com.ssdd.cs.client.CriticalSectionRequestConcurrentSender*, con el cual, se envían de forma concurrente todas las peticiones necesarias mediante un *threadpool*.

Tras esto, en cada uno de los procesos solicitados, en el endpoint */request*, se realizan las operaciones pertinentes, y en el caso de que el acceso sea permitido, se responde a la petición *HTTP* con *status* 204 (la petición ha finalizado de forma satisfactoria).

En el caso de que el acceso no sea permitido en este momento y el proceso requiera ser “encolado”, se obtiene el *CriticalSectionQueue* asociado al proceso solicitado. Tras ello, se invoca a el método *waitInQueue*, donde el hilo creado para atender la petición al *endpoint* */request*, permanecerá bloqueado, hasta que el proceso solicitado, realice una petición a el *endpoint /release* (ya que eso significará que se libera la sección crítica y por tanto se le concede acceso a la petición). Una vez terminado este proceso, se responderá a la petición *HTTP* con *status* 204.

Volviendo a el lado del cliente, para considerar que se tiene acceso a la sección critica, lo que se hará será esperar a que todas las peticiones sean respondidas.

Para implementar esto, lo que se hace es esperar a que todas las tareas de la *threadpool* (que consisten todas en envió de una petición y esperar a que esta sea respondida) acaben.

\* anotación: las peticiones se hacen a cada proceso y no a cada servidor. Esto a pesar de utilizar más mensajes de petición, hace que se pueda solicitar solo a los procesos deseados, pudiendo configurar más de una sección critica de forma concurrente en el mismo servicio. También, permite mantener secciones criticas solapadas pasándole correctamente los parámetros de ejecución a los distintos procesos.

## Gestión de “estado de proceso” en el servicio de sección critica

Cada vez que se realiza un acceso a un *endpoint* en el que se necesita acceder o modificar el estado de un proceso, se llama a el método *com.ssdd.cs.service.CriticalSectionService#getProcessState*, el cual a partir del id de un proceso devuelve un objeto *CriticalSectionProcessState*, que contiene el estado del proceso.

En el caso de no encontrar el estado del proceso, lanza una excepción del tipo *ProcessNotFoundException*, la cual está preparada para devolver un código de error 404 a la petición.

Una vez obtenido el estado del proceso, para mantener la consistencia sobre este, se trabaja por transacciones sobre el estado de cada proceso. De esta forma, cuando se desea comenzar a realizar operaciones sobre el estado de un proceso, se llama a el método *lock*, el cual tratara de hacer una operacion *acquire* sobre un *java.util.concurrent.Semaphore* que mantienen como atributo el *CriticalSectionProcessState*, bloqueando cualquier operación sobre este.

Una vez se ha terminado de realizar operaciones sobre el “estado de proceso”, esto hace que se llame a el método *unlock*, el cual realizara una operación *release* sobre el semáforo ya mencionado. Esto “terminara la transacción”, permitiendo a otros hilos trabajar sobre el ”estado de proceso”.

De primeras, esto puede parecer sobresincronización, no obstante, hay fragmentos de código, como el establecer que se desea acceder en el que hay que tomar el valor del contador de *Lamport* del proceso, cambiar el estado de la sección critica para el proceso y activar el encolado de forma atómica, lo cual hace necesario algún mecanismo para asegurar la consistencia durante las transacciones.

## “Encolado” de procesos en la sección critica

El encolado de los procesos, a primera vista parece trivial, pues con un semáforo, se podría implementar, pero ese mecanismo no es consistente con el resto de las sincronizaciones y produce un interbloqueo.

El proceso de petición de entrada a la sección critica se define en el *enpoint /request*, cuyo pseudocódigo se aprecia en la figura 3.5.1.

*state = getProcess(id);  
state.lock();*

*state.updateLamport(ti);  
 boolean wait = state.permitEnter();  
state.unlock();  
if(wait)  
 state.getQueue().waitQueue();*

Figura 3.5.1

Aquí surge un problema, y es que la espera en la cola si corresponde, se debería hacer antes de realizar la operación *unlock* para hacerse de forma atómica, pero esto provocaría un bloqueo del sistema, debido a que el propio proceso solicitado también tiene que realizar una operación *lock* para comenzar su transacción sobre su estado en el *enpoint* */release* para liberar a el proceso encolado. Cosa que no puede ocurrir si el proceso que solicita no cierra antes su transacción con *unlock*.

La primera solución es establecer la espera fuera de la sección critica como se ve en el pseudocódigo, pero esto puede producir un interbloqueo, porque si el proceso preguntado libera a los procesos esperando en la cola entre la operación *unlock* y la operación *waitQueue*, el proceso que espera por la respuesta se quedará esperando a la respuesta del servidor (porque el hilo que atiende la petición en el endpoint /request estará bloqueado).

Esto provocara que la siguiente vez que el proceso preguntado quiera realizar un acceso a la sección critica tenga una marca de tiempo superior a la de el proceso que le pregunto antes (ahora bloqueado) y por tanto que este en el proceso de petición lo ponga a esperar en la cola. Esto provocara que al encolarse mutuamente se de un interbloqueo, que se propagara a todo el sistema, haciendo que todos los procesos esperen por el primer proceso que quedó bloqueado.

La solución final planteada para este problema fue establecer una clase *CriticalSectionQueue*, que tiene dos atributos:

* *queueingAllowed*: establecido a true en el caso de que sea permitido encolar y a false en caso contrario.
* *waitingProcesses*: una lista de *java.util.concurrent.Semaphore*.

De esta forma, el mecanismo establecido en esta clase es el siguiente:

* Cuando un proceso llama a su *endpoint /set/requested*, dentro de la transacción que abre sobre su estado, llamara a el método actívate (el cual es *synchronized*, lo que permite a un único hilo entrar en dicho método), en donde se establecerá a *true* el atributo *queueingAllowed*.
* Cuando un proceso desee encolarse, invoca el método *waitInQueue*, con el cual, se entra en un fragmento sincronizado con la clase (delimitado por *synchronized(this)*), en el cual, si está habilitado el “encolamiento”, instancia un *java.util.concurrent.Semaphore* con 0 *permits* y lo añade a la lista *waitingProcesses*.

Una vez hecho esto, sale del fragmento de código sincronizado y espera por el semáforo en el caso de que este haya sido creado (si no su valor será *null*).

* Por último, cuando un proceso llama a su *endpoint /release*, dentro de la transacción que abre sobre su estado, llamara a el método *deactivateAndrelease* (el cual es *synchronized*, lo que permite a un único hilo entrar en dicho método), en donde se establecerá a *false* el atributo *queueIngAllowed*, se efectúa una operación *release* sobre cada semáforo que se encuentre en *waitingProcesses* y se vacía esta lista.

De esta forma, aunque el “encolarse” se realice fuera de la transacción, se soluciona el problema debido a que se evitan las siguientes casuísticas:

* Si cuando va a encolarse ya se ha liberado la sección critica, *queueingAllowed* tendrá el valor *false,* y este no esperará porque no se instanciará un semáforo.
* Si cuando se realice la operación *acquire* sobre el semáforo ya se ha liberado la sección critica, este tendrá un *permit* y el hilo no tendrá que esperar.

*FAQ* sobre este mecanismo:

* Este mecanismo de espera no puede ser un *lock* (del tipo *Object*) implementado con *synchronized*, *wait* y *notifyAll*, porque se necesita un objeto con *permits* que permanezcan en el tiempo y no únicamente en el instante en el que se realiza la liberación, como se da en el caso de *notifyAll*.
* Tampoco se puede implementar con un único semáforo al que se le otorguen tantos *permits* como procesos esperando cuando se realice el *deactivateAndRelease*, debido a que dichos *permits* podrían tomarlos otros procesos que no son los que deseamos.
* Tampoco se puede implementar con una barrera del tipo *java.util.concurrent.CyclicBarrier*, porque tendríamos que conocer de antemano cuantos procesos van a esperar, lo cual es desconocido, porque pueden ser de 0 a N-1 (siendo N el número de proceso del sistema).

## Problemas en el desarrollo

En cuanto a la sincronización, cabe destacar que, durante el desarrollo, el principal problema en materia de sincronización consistió en que no utilizábamos bien las marcas de tiempo, confundiendo la marca de tiempo de la última petición, con el contador de *Lamport* del proceso, resultando en un comportamiento anómalo del sistema (figura 3.6.2).

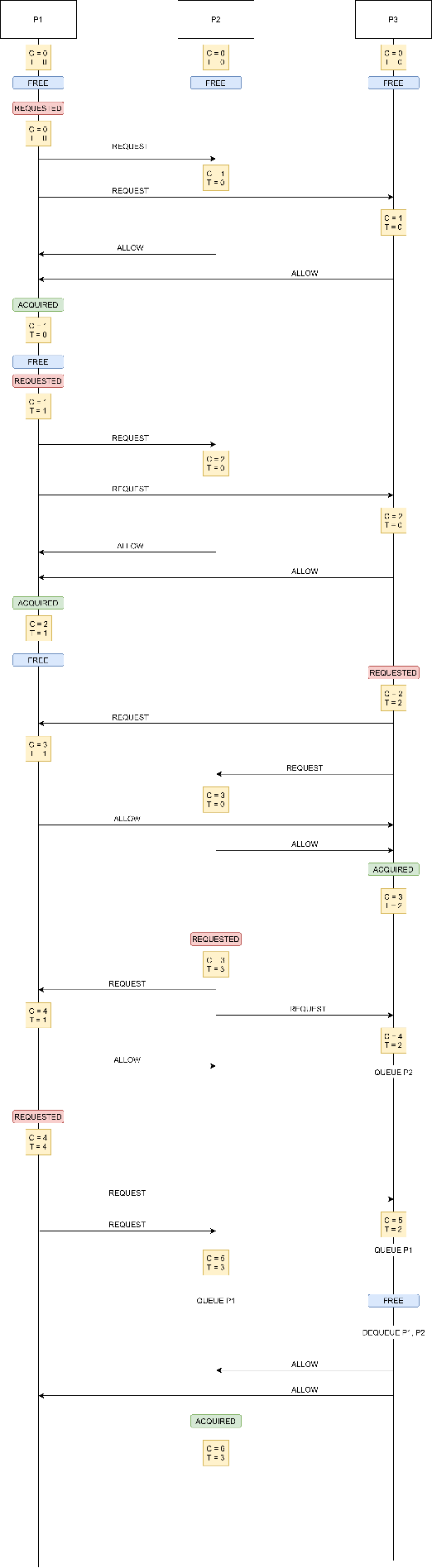
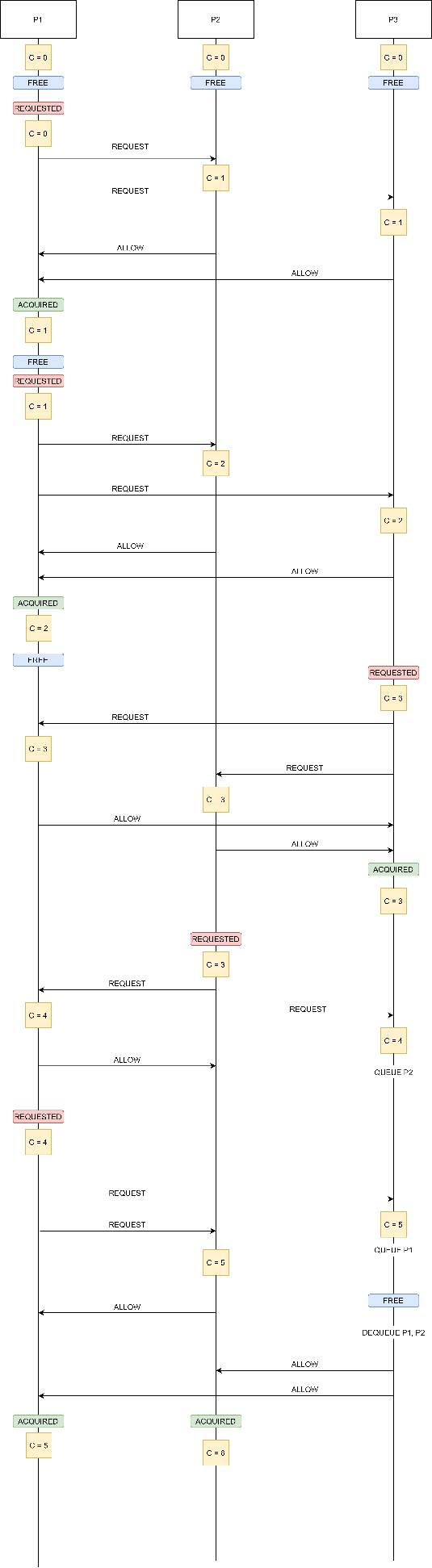


Figura 3.6.1: Figura 3.6.2:

Funcionamiento correcto del sistema Funcionamiento anómalo del sistema

# Aspectos relevantes de NTP

En cuanto al algoritmo utilizado para seleccionar el mejor par, se ha utilizado *Marzullo*.

La configuración de NTP, como ya se ha mencionado antes, se encuentra en la clase *com.ssdd.util.constants.INtpConstants*. No obstante, de esta cabe destacar, que se puede configurar el tiempo de espera entre muestreos de tiempo en el servicio de *NTP* y si se va a realizar este o no.

De NTP cabe destacar, que, para realizar el proceso de petición de tiempos con varios servicios, se utiliza un *com.ssdd.util.concurrent.ConcurrentExecutor*, que realiza este proceso, de manera concurrente, para tardar únicamente en lo que se tardaría en muestrear un servicio.

Por último, es destacable, que el formato en el que se devuelve el array de tiempos del servidor es *JSON*, para lo cual se utiliza para serializar y deserializar la biblioteca *GSON*.

# Explicación del contenido de los paquetes

En este apartado se procederá a explicar el contenido y funcionalidad de cada paquete junto a las clases principales de este. Cabe destacar que se explicarán aquellas funciones cuya funcionalidad no sea trivial para facilitar su comprensión, ya que cualquier otro detalle se puede consultar en el *Javadoc* del proyecto, situado en la carpeta *doc*.

## Paquete com.ssdd.cs.bean

Este paquete contiene los *POJO* utilizados para la gestión de la sección critica mediante el algoritmo de *Ricart* y *Argawala*.

### CritricalSectionState.java

Enumeración que contiene los tres estados en los que se puede encontrar un proceso para su acceso a la sección crítica:

* ***FREE***: estado en el cual el proceso no quiere entrar en la sección crítica.
* ***REQUESTED***: estado en el cual el proceso quiere entrar en la sección crítica.
* ***ACQUIRED***: estado en el cual el proceso se encuentra en la sección crítica.

### LamportCounter.java

Contiene las funciones necesarias para la representación de un reloj de Lamport. Es utilizado por los servicios, para representar el estado de un proceso. Entre sus funciones se encuentra:

* **Método *update***: incrementa el valor del contador de la siguiente forma:
  + *LC1*: utiliza esta fórmula para incrementar en una unidad si no recibe argumentos.
  + *LC2*: utiliza esta fórmula si recibe el valor de otro contador como argumento e incrementa de la siguiente forma:
    - Si el valor del argumento es menor que el valor del contador, entonces se incrementa el contador.
    - En caso contrario al anteriormente relatado, se signa como nuevo valor del contador el del argumento incrementado en una unidad.

## Paquete com.ssdd.cs.client

Este paquete contiene la clase cliente del servicio de la sección crítica junto a la clase necesaria para proveer el acceso a un proceso mediante su servicio asociado.

### CritricalSectionClient.java

Clase cliente del servicio *CriticalSectionService*. Este cliente contiene como campos un identificador (del proceso en el sistema), el *CriticalSectionRouter* y la lista de procesos que intentan acceder a la sección crítica (excepto el propio proceso).

### CritricalSectionRouter.java

Esta clase es la encargada de encaminar a partir de un identificador de u n proceso, a el servicio asociado a este. Su funcionamiento es el siguiente:

* Cuando se solicita un servicio, lo busca entre los identificadores que conoce.
* En caso de no encontrarlo, realizar una petición (a cada servicio del sistema) para obtener los procesos asociados a cada servicio. Posteriormente, devuelve el proxy al servicio al cual está asociado proceso que se buscaba
* En caso de encontrarlo, retorna directamente el servicio asociado al proceso cuyo identificador se ha pasado como argumento.

Entre las funciones que usa, se encuentran:

* **Método *route***: lleva a cabo el proceso de retorno del servicio asociado al proceso.
* **Método *update***: actualiza la lista de asociaciones locales con nueva información de asociaciones aportada por parámetro.
* **Método *updateAll***: lleva a cabo el proceso de solicitud de servicios de cada proceso subscrito.

## Paquete com.ssdd.cs.main

Este paquete contiene las clases necesarias para la ejecución de las distintas partes de la simulación.

### Main.java

Esta clase es la principal de la ejecución y se encargará de ir llamando a las distintas clases principales correspondientes a las partes en las cuales se ha dividido la ejecución. Estas clases se explican a continuación.

### MainLogVerification.java

Esta clase es la encargada de comprobar si en el log de acceso a la sección crítica se mantiene la consistencia de esta. Lleva a cabo la ejecución de Comprobador para terminar de verificar que no se han realizado violaciones de la sección crítica.

### MainSimulation.java

Esta clase llevará a cabo la construcción e iniciación de los procesos que participan la ejecución.

El rango de procesos a gestionar se le indica por parámetro.

### MainSupervisor.java

Esta clase es la encargada de realizar varias tareas.

* Toma de muestras de *NTP*
* Corrección de tiempos ficheros de log de sección critica
* Reinicio de la sección critica.

## Paquete com.ssdd.ntp.client

Este paquete contiene la clase que implementa el cliente *NTP*, para realizar los muestreos de tiempo, cálculo de pares (*delay, offset*) y selección del mejor par mediante el algoritmo de *Marzullo*.

## Paquete com.ssdd.simulation

Este paquete contiene las clases necesarias para llevar a cabo la simulación de accesos a la sección critica, el *logging* de estos y la corrección de tiempos de este *log*.

### SimulationProcess.java

Lleva a cabo la simulación

### SimulationProcessBuilder

Patrón *builder*, para construir objetos del tipo *SimulationProcess*.

### SimulationLog.java

Esta clase escribe el log con el formato requerido por el enunciado. Refleja las entradas y salidas de la sección crítica para cada uno de los procesos junto a su tiempo en un timestamp medido en milisegundos.

### SimulationLogAdjuster.java

Esta clase es la encargada de ajustar los tiempos representados en el log (para cada acceso y salida de la sección crítica de cada uno de los procesos que no se encuentre en la máquina de referencia) respecto al offset calculado mediante NTP para cada servicio.

# Despliegue

Para el despliegue, se necesita tener instalado java 1.8\_241, para lo cual, en el entorno Ubuntu en el que hemos realizado el desarrollo, se instala mediante los siguientes comandos:

*sudo apt update*

*sudo apt install openjdk-8-jdk openjdk-8-jre*

También se requiere que el puerto 8080 este disponible, porque es donde corre la apliacion, no obstante, si se configura *Apache Tomcat 7.0.103* y se cambia el valor de *com.ssdd.util.constants.IConstants#BASE\_URI\_FORMAT*, se puede cambiar el puerto en el que corre esta.

Tras lo cual, se puede comenzar el proceso de despliegue:

1. situarse en la carpeta *deploy*
2. generar certificado RSA (si no se tiene uno disponible) y enviarlo a las maquinas donde se ejecutará la aplicación:

*bash deployer.bash -genkey -newsession user host1 -newsession user host2 -newsession user host3*

1. Instalar servicio de *Tomcat 7.0.103*\* y los ejecutables (*.war* y *.jar*) con el siguiente comando:

*bash deployer.bash -deploy user host1 host2 host3*

Nota: es necesario que el usuario sea *root* para realizar esta operación, ya que se necesita activar un servicio.

1. Ejecutar la simulación, tratamiento de logs y verificación de estos con el siguiente comando:

*bash deployer.bash -run true user host1 host2 host3*

Nota: si en vez de true se establece a false, no se realizará muestreo de NTP ni corrección de logs. Lo cual es útil para realizar una simulación en local.

1. Opcionalmente se puede limpiar todo el despliegue (excepto el servicio de *Tomcat*) con el siguiente comando:

*bash deployer.bash -clean user host1 host2 host3*

Cabe destacar que se tiene una última operación, *redeploy,* que permite desplegar de nuevo los ejecutables de la aplicación (*.jar* y *.war*).

Por último, cabe destacar que se puede dar un problema a la hora de desplegar el servidor, y es que este script está pensado para desplegar Apache Tomcat 7.0.103, que es el ofrecido en el enlace proporcionado por *Apache Foundation* para descargar *Apache Tomcat 7*.

Debido a esto, si se actualiza este enlace o bien se actualiza la versión de este, el despliegue de *Tomcat 7* dejará de estar operativo, pero podrá ser solucionado comentado la parte de descargar *Tomcat* del *script* e indicando la ruta donde esta la carpeta de instalación de este para poder ser copiada a las maquinas donde se ejecutará la práctica.

Nota: se adjunta otro script además de *deployer.bash*, el cual se llama *test\_multiple.bash*, el cual permite realizar múltiples ejecuciones de *deployer.bash* seguidas y visualizar los resultados de estas. Ademas sirve como ejemplo de uso de *deployer.bash.*