Sistemas Distribuidos: Práctica de Semáforos Distribuidos (DMUTEX)

SD #DMUTEX'12

José María Peña Sánchez

Índice

1. Introducción

Esta práctica plantea el desarrollo de un escenario esperimental que implemente un algoritmo de exclusión mutua distribuida. En concreto se plantea la implementación de un algoritmo basado en marcas de tiempo. De esta forma la práctica se divide en dos partes:

① Diseño de un sistema de relojes lógicos vectoriales.

② Implementación del algoritmo de exclusión mutua distribuida.

El prototipo que se va a desarrollar es un modelo muy simplificado que no tendrá que considerár ningún caso de fallo o caida de procesos, detección de interbloqueos o situaciones anómalas similares.

1.1. Relojes Lógicos Vectoriales

Muchos algoritmos necesitan establecer unos criterios de tiempo entre los diferentes procesos distribuidos que los componen. Como alternativa a la sincronización por medio de tiempo real, Lamport desarrolló un sistema de contadores, denominados relojes lógicos (LC) basado en relaciones de precedencia. El esquema que se muestra a continuación es una extensión de dicho sistema a relojes lógicos vectoriales.

El sistema está compuesto por N procesos fijos. Cada proceso i del sistema dispone de un vector de valores denominado reloj lógico, LC_i. Este vector tiene tantos elementos como procesos, es decir N. Cada componente del vector está asociado al proceso del mismo índice.

Los contadores de relojes lógicos avanzan segun se producen *eventos*. Un evento es tanto una acción interna como el envio o la recepción de un mensaje:

- ❒ Cuando un evento se produce en el componente i, entonces:

$$LC_i[i] = LC_i[i] + 1 \tag{1}$$

- ❒ Si el componente i envia un mensaje a cualquier otro componente, esto se considera un evento y además, junto al mensaje, se transmite el vector de valores del reloj lógico de i, LC_i.

  `Mensaje : {datos} + {LC}`

- ❒ Si el componente j recibe un mensaje del componente i, entonces dispone del reloj lógico del emisor que combina con su propio reloj lógico actualizando este último:

$$\forall k : LC_j[k] = max(LC_i[k], LC_j[k]) \tag{2}$$
$$LC_j[j] = LC_j[j] + 1 \tag{3}$$

  La recepción de un mensaje también es un evento y por lo tanto actualiza el contador (ver fórmula 3)).

El uso de los relojes lógicos permite definir una relación de orden parcial entre los sucesos. Si cada suceso tiene un instante asociado (representado por un valor del vector del reloj lógico del proceso donde se produjo), la comparación entre dichos sucesos es la siguiente: Sean LC_A y LC_B los relojes lógicos de los eventos A y B respectivamente:

❶ El evento A es anterior a B si y sólo si:

$$\forall k : LC_A[k] < LC_B[k] \tag{4}$$

❷ El evento B es anterior a A si y sólo si:

$$\forall k : LC_B[k] < LC_A[k] \tag{5}$$

❸ En el resto de casos la anterioridad de los eventos A y B no puede ser determinada.

Para hacer que esta relación de orden parcial sea de orden total es necesario resolver el caso ❸. Si en dichas circunstancias se considera anterior el evento asociado al proceso con **el identificador más bajo**, entonces toda pareja de eventos puede ser comparada.

1.2. Exclusión Mútua Distribuida

Dado un conjunto de n procesos $P_1, P_2, P_3, ...P_n$, el mecanismo para acceder a una región de exclusión mutua consiste en los siguientes pasos.

① Si un proceso P_i desea hacer un `LOCK` para acceder de forma exclusiva a la región, manda al resto de procesos un mensaje `LOCK_MESSAGE` junto al valor de su reloj lógico LC_i. Dicho mensaje se deberá transmitir a varios destinatarios de la forma más eficiente posible (si existe mecanismos multicast, lo cual implicaría sólo un envío).

② Todo proceso que recibe un mensaje `LOCK_MESSAGE`:

- ❒ Si el proceso receptor no desea, a su vez, entrar en la misma región crítica entonces devuelve un mensaje `OK_MESSAGE`.
- ❒ Si el proceso se encuentra dentro de la región crítica entonces no responderá nigún mensaje.
- ❒ Si el proceso receptor sí desea entrar en la misma región crítica entonces se comparan los instantes de petición de cada uno de los accesos:
  - ❍ Si el mensaje lleva un reloj lógico anterior al instante en que el proceso solicito el acceso a la región, éste responde con un `OK_MESSAGE`.
  - ❍ Si el instante en el que el proceso solicito el acceso a la región es anterior al del mensaje recibido entonces no se responderá a dicha petición.

③ Un proceso accede a la región crítica despues de haber solicitado el acceso en el momento en el que recopila $n-1$ respuestas de tipo `OK_MESSAGE`.

④ Al salir de una región crítica el proceso enviará los mensajes `OK_MESSAGE` que le quedasen pendientes a los procesos solicitantes a los que no hubiese respondido.

NOTA: Hay que tener en cuenta que en los casos en los que dos procesos compiten para el acceso a la región crítica se deben considerar los siguientes aspectos:

- ✔ La comparación entre relojes lógicos debe ser completa, es decir que en el caso de que los eventos no sean distingibles se usará en índice de proceso como discriminante.
- ✔ Los relojes lógicos a compara por parte de un proceso son, por un lado el recibido en el mensaje y por el otro el del instante en el que él realizo la petición, **no el instante actual del proceso**.

2. Desarrollo de la Práctica

La práctica consiste en el desarrollo de un prototipo que implemente la lógica de control para la gestión de relojes lógicos vectoriales así como el protocolo de acceso a una región con exclusión mutua. Para ello se deberá desarrollar un programa que:

- ❒ Mediante la entrada estándar (`stdin`) reciba la secuencia de acciones a realizar.
- ❒ Utilice el protocolo UDP y la librería de *sockets* para comunicarse con otros procesos análogos.
- ❒ Genere mensajes de traza y control por medio de la salida estándar (`stdout`).

El programa solicitado sólo tendrá que tener un único hilo de ejecución (no es necesario implementar varios *threads*) puesto que la entrada de acciones será la que determinará la secuencia de sucesos que atenderá el proceso. Se ha de entender que esto es una implementación de un prototipo y que este mecanismo permite forzar secuencias de acciones que permitan estudiar el comportamiento de ambos algoritmos (es una simulación).

2.1. Entradas y Salidas del Proceso

Cada proceso recibirá órdenes e información por medio de la entrada estándar de la misma forma usará su salida estándar para notificar determinadas trazas de ejecución.

2.1.1. Inicialización del Proceso: Cada proceso recibe como argumento de entrada el nombre simbólico que usará durante la ejecución:

```
$ proceso identificador
```

Nada más arrancar el proceso reservará un puerto UDP y abrirá un *socket* asociado al mismo. Para ello buscará un puerto libre cualquiera y notificará por medio de la salida estándar su identificador así como el puerto reservado, con formato `"%s:%d\n"`:

identificador: *número_puerto*

Despues de eso el proceso recibirá por la entrada estándar líneas con el mismo formato que identificarán otros procesos y sus puertos de acceso. El número de líneas (y por lo tanto de procesos) que se le pueden notificar será variable. De dichos procesos deberá identificar, de cara a futuras comunicaciones su identificador simbólico y el puerto asociado. Dos procesos **sólo** pueden comunicarse por medio de mensajes UDP.

Todos los procesos del sistema recibirá en el mismo orden la información de todos los procesos del sistema, recibiendo además su propia información. Es decir que si el sistema tiene 3 procesos A, B y C, todos los procesos recibiran en el mismo orden A, B y C junto a sus

puertos respectivos.

La definición de otros procesos concluye con un mensaje `"START\n"` por la entrada estándar. En ese momento comienza el bucle de procesamiento de acciones, la simulación propiamente icha.

2.1.2. Acciones: Tras la inicialización el proceso recibira de forma externa (por la entrada estándar) una serie de acciones a realizar. Cada acción vendrá representada por una y sólo una línea. Como fruto de alguna de estas acciones el proceso generará trazas que serán escritas en la salida estándar. Toas las trazas emitidas tendrán el formato:

identificador: *traza*

Las posibles acciones que deberá interpretar el servidor son:

- `EVENT`: Implica un nuevo evento producido dentro del proceso. Esto implica que se deberán registrar dicho suceso en el componente apropiado del reloj vectorial y que se emitirá una traza `"TICK"`.
- `GETCLOCK`: Esta acción es una acción de monitorización que no tiene que implicar ninguna avance en el reloj lógico del proceso. El objeto de esta acción es hacer que el proceso emita una traza con su reloj lógico. El formato de la traza será `"LC[`x_1`,`x_2`,`x_3`]"` (para el caso e un sistema con tres procesos).
- `MESSAGETO` *id*: Por medio de esta acción se pide al proceso que mande un mensaje al proceso *id*. El mensaje será un mensaje vacío que sólo contendrá la información del reloj lógico. Esta acción generá dos mensajes de traza, en primer lugar un `"TICK"` (por tratarse de un evento) y justo despues una traza `"SEND(MSG,`*id*`)"`.
- `RECEIVE`: Esta acción solicita al proceso que reciba el primer mensaje UDP que tenga pendiente de recoger del *socket* UDP. La traza generada, al igual que en el caso anterior será doble, en primer lugar `"RECEIVE(`*tipo*`,`*id*`)"`, donde *tipo* es el tipo de mensaje recibido[1] e *id* es el proceso emisor del mensaje y una segunda traza `"TICK"` (al ser la recepción de un mensaje otro evento).
- `LOCK` *sec*: Esta acción indica que el proceso en cuestión desea entrar dentro de la sección crítica *sec*. Ésta se tratará como lo indica el algoritmo de exclusión mútua indicado anteriormente, es decir emitiendo todos los mensajes de tipo `LOCK` al resto de procesos del sistema. Se debe considerar que existe un mecanismo eficiente para transmitir un mensaje a varios destinatarios, por lo tanto este envío a varios receptores se considerará un **único evento**, a nivel del contador del reloj lógico. Todos los envios y el único `TICK` asociados serán registrados como mensajes de traza.

[1] Tipos de mensajes: `MSG`, `LOCK`, `OK`

- ❒ `UNLOCK sec`: Esta acción implica que se desea salir de una sección crítica antes conseguida. Esta acción puede implicar el envio de mensajes de tipo `OK` a otros procesos que esperan acceso a la región crítica o si no hay ninguno no generará traza alguna.

Las trazas realizadas por las acciones `LOCK` y `UNLOCK` serán las mismas que las que generarían los envios de mensajes correspondientes.

NOTA: Hay un caso especial de traza que hay que considerar en las acciones `RECEIVE`, si se trata de un mensaje de tipo `OK` y que además es el último que estaba esperando el proceso (mensaje número $n-1$) para entrar en dicha sección, entonces emitirá una traza `"MUTEX(`*sec*`)"` adicional, para indicar que ha entrado en la sección crítica.

2.2. Código de Apoyo

El código de apoyo se encuentra dividido en tres directorios:

- ❒ `Controlador`: Este directorio contiene el código del programa `controlador` de pruebas. Este código **NO DEBE SER MODIFICADO**. El programa controlador es el encargado de ejecutar varios procesos, proporcionarles acciones y recopilar los resultados de trazas.

- ❒ `Proceso`: Este directorio es donde se encuentra el código del proceso que debe implementar los algoritmos propuestos. Se proporciona un pequeño esqueleto de la función principal como sugerencia de la estructura de este proceso. Este directorio es **EL ÚNICO QUE SE RECOGERÁ EN LA ENTREGA DE LA PRÁCTICA**.

- ❒ `Ejemplos`: Este directorio contiene varios ficheros de ejemplo usados por el programa `controlador`. Los resultados de estos ejemplos se muestran en la siguiente sección.

El código de apoyo se podrá encontrar en la página web de la asignatura:

`http://laurel.datsi.fi.upm.es/docencia/asignaturas/sod`

3. Ejemplos de Ejecución

Existen dos formas de verificar el funcionamiento de los procesos, por un lado se pueden arrancar dichos procesos en diferentes ventanas y se interactúa con ellos por medio de la entrada/salida estándar. Como alternativa se puede usar el programa controlador que lee un fichero de ordenes y, posteriormente, arranca y se comunica con el número de procesos necesarios.

3.1. Ejemplo de Individual

Como muestra del funcionamiento de un proceso se presenta a continuación la forma de ejecutarlo de forma individual. Este ejemplo hace que dicho proceso se comunique con él mismo y se intercambie ciertos mensajes. Con la notación siguiente se muestra `la salida de dicho proceso`. Y con la notación `la entrada del mismo, es decir lo tecleado por el usuario`.

```
[chema@quino ~/DATSI/SD/DMUTEX.2012]./proceso P
P: 1188
P: 1188
START
GETCLOCK
P: LC[0]
EVENT
P: TICK
GETCLOCK
P: LC[1]
MESSAGETO P
P: TICK
P: SEND(MSG,P)
RECEIVE
P: RECEIVE(MSG,P)
P: TICK
GETCLOCK
P: LC[3]
FINISH
[chema@quino ~/DATSI/SD/DMUTEX.2012]
```

3.2. Ejemplo con Controlador

3.2.1. Ejemplos/01.ord:

```
[chema@quino ~/DATSI/SD/DMUTEX.2012]./controlador Ejemplos/01.ord
PROCESO: A: 1188
PROCESO: B: 1189
PROCESO: C: 1190
# Ejemplo sencillo de ordenes
# Procesos: A, B, C
A: [EVENT]-> A{TICK} B{--} C{--}
B: [EVENT]-> A{--} B{TICK} C{--}
C: [EVENT]-> A{--} B{--} C{TICK}
A: [GETCLOCK]-> A{LC[1,0,0]} B{--} C{--}
B: [GETCLOCK]-> A{--} B{LC[0,1,0]} C{--}
```

```
C: [GETCLOCK]-> A{--} B{--} C{LC[0,0,1]}
FINISH[14876]
FINISH[14877]
FINISH[14878]
```

3.2.2. Ejemplos/02.ord:

```
[chema@quino ~/DATSI/SD/DMUTEX.2012]./controlador Ejemplos/02.ord
PROCESO: A: 1190
PROCESO: B: 1191
PROCESO: C: 1192
# Ejemplo sencillo de ordenes y mensajes
# Procesos: A, B, C
A: [EVENT]-> A{TICK} B{--} C{--}
B: [EVENT]-> A{--} B{TICK} C{--}
C: [EVENT]-> A{--} B{--} C{TICK}
A: [MESSAGETO B]-> A{TICK|SEND(MSG,B)} B{--} C{--}
B: [EVENT]-> A{--} B{TICK} C{--}
B: [RECEIVE]-> A{--} B{RECEIVE(MSG,A)|TICK} C{--}
A: [MESSAGETO C]-> A{TICK|SEND(MSG,C)} B{--} C{--}
C: [RECEIVE]-> A{--} B{--} C{RECEIVE(MSG,A)|TICK}
C: [MESSAGETO A]-> A{--} B{--} C{TICK|SEND(MSG,A)}
B: [MESSAGETO A]-> A{--} B{TICK|SEND(MSG,A)} C{--}
A: [RECEIVE]-> A{RECEIVE(MSG,C)|TICK} B{--} C{--}
A: [GETCLOCK]-> A{LC[4,0,3]} B{--} C{--}
A: [RECEIVE]-> A{RECEIVE(MSG,B)|TICK} B{--} C{--}
A: [GETCLOCK]-> A{LC[5,4,3]} B{--} C{--}
B: [GETCLOCK]-> A{--} B{LC[2,4,0]} C{--}
C: [GETCLOCK]-> A{--} B{--} C{LC[3,0,3]}
FINISH[14885]
FINISH[14886]
FINISH[14887]
```

3.2.3. Ejemplos/03.ord:

```
[chema@quino ~/DATSI/SD/DMUTEX.2012]./controlador Ejemplos/03.ord
PROCESO: P0: 1192
PROCESO: P1: 1193
# Ejemplo de un semaforo entre dos procesos
# Procesos: P0,p1
P0: [GETCLOCK]-> P0{LC[0,0]} P1{--}
P1: [GETCLOCK]-> P0{--} P1{LC[0,0]}
```

```
P0: [LOCK S]-> P0{TICK|SEND(LOCK,P1)} P1{--}
P1: [RECEIVE]-> P0{--} P1{RECEIVE(LOCK,P0)|TICK|TICK|SEND(OK,P0)}
P1: [EVENT]-> P0{--} P1{TICK}
P0: [RECEIVE]-> P0{RECEIVE(OK,P1)|TICK|MUTEX(S)} P1{--}
P0: [UNLOCK S]-> P0{--} P1{--}
P1: [EVENT]-> P0{--} P1{TICK}
P0: [GETCLOCK]-> P0{LC[2,2]} P1{--}
P1: [GETCLOCK]-> P0{--} P1{LC[1,4]}
FINISH[14896]
FINISH[14897]
```

3.2.4. Ejemplos/04.ord:

```
[chema@quino ~/DATSI/SD/DMUTEX.2012]./controlador Ejemplos/04.ord
PROCESO: P0: 1193
PROCESO: P1: 1194
# Ejemplo de un semaforo entre dos procesos (bloqueo).
# Procesos: P0,p1
P0: [GETCLOCK]-> P0{LC[0,0]} P1{--}
P1: [GETCLOCK]-> P0{--} P1{LC[0,0]}
P0: [LOCK S]-> P0{TICK|SEND(LOCK,P1)} P1{--}
P1: [LOCK S]-> P0{--} P1{TICK|SEND(LOCK,P0)}
P1: [RECEIVE]-> P0{--} P1{RECEIVE(LOCK,P0)|TICK|TICK|SEND(OK,P0)}
P1: [EVENT]-> P0{--} P1{TICK}
P0: [GETCLOCK]-> P0{LC[1,0]} P1{--}
P0: [RECEIVE]-> P0{RECEIVE(LOCK,P1)|TICK} P1{--}
P0: [RECEIVE]-> P0{RECEIVE(OK,P1)|TICK|MUTEX(S)} P1{--}
P0: [UNLOCK S]-> P0{TICK|SEND(OK,P1)} P1{--}
P1: [RECEIVE]-> P0{--} P1{RECEIVE(OK,P0)|TICK|MUTEX(S)}
P1: [EVENT]-> P0{--} P1{TICK}
P1: [UNLOCK S]-> P0{--} P1{--}
P0: [GETCLOCK]-> P0{LC[4,3]} P1{--}
P1: [GETCLOCK]-> P0{--} P1{LC[4,6]}
FINISH[14901]
FINISH[14902]
```

3.2.5. Ejemplos/05.ord:

```
[chema@quino ~/DATSI/SD/DMUTEX.2012]./controlador Ejemplos/05.ord
PROCESO: A: 1194
PROCESO: B: 1195
PROCESO: C: 1196
```

```
PROCESO: D: 1197
# Ejemplo con multiples semaforos
# Procesos: A,B,C,D
# Semaforos: X,Y
A: [EVENT]-> A{TICK} B{--} C{--} D{--}
B: [EVENT]-> A{--} B{TICK} C{--} D{--}
B: [EVENT]-> A{--} B{TICK} C{--} D{--}
C: [EVENT]-> A{--} B{--} C{TICK} D{--}
D: [EVENT]-> A{--} B{--} C{--} D{TICK}
A: [GETCLOCK]-> A{LC[1,0,0,0]} B{--} C{--} D{--}
B: [GETCLOCK]-> A{--} B{LC[0,2,0,0]} C{--} D{--}
C: [GETCLOCK]-> A{--} B{--} C{LC[0,0,1,0]} D{--}
D: [GETCLOCK]-> A{--} B{--} C{--} D{LC[0,0,0,1]}
# Los procesos A y B intentan cerrar X e Y, respectivamente
A: [LOCK X]-> A{TICK|SEND(LOCK,B)|SEND(LOCK,C)|SEND(LOCK,D)} B{--} C{--} D{--}
B: [LOCK Y]-> A{--} B{TICK|SEND(LOCK,A)|SEND(LOCK,C)|SEND(LOCK,D)} C{--} D{--}
# Todos reciben las peticiones
B: [RECEIVE]-> A{--} B{RECEIVE(LOCK,A)|TICK|TICK|SEND(OK,A)} C{--} D{--}
C: [RECEIVE]-> A{--} B{--} C{RECEIVE(LOCK,A)|TICK|TICK|SEND(OK,A)} D{--}
D: [RECEIVE]-> A{--} B{--} C{--} D{RECEIVE(LOCK,A)|TICK|TICK|SEND(OK,A)}
A: [RECEIVE]-> A{RECEIVE(LOCK,B)|TICK|TICK|SEND(OK,B)} B{--} C{--} D{--}
C: [RECEIVE]-> A{--} B{--} C{RECEIVE(LOCK,B)|TICK|TICK|SEND(OK,B)} D{--}
D: [RECEIVE]-> A{--} B{--} C{--} D{RECEIVE(LOCK,B)|TICK|TICK|SEND(OK,B)}
# Tanto A como B deben recibir los 3 OKs
A: [RECEIVE]-> A{RECEIVE(OK,B)|TICK} B{--} C{--} D{--}
A: [RECEIVE]-> A{RECEIVE(OK,C)|TICK} B{--} C{--} D{--}
A: [RECEIVE]-> A{RECEIVE(OK,D)|TICK|MUTEX(X)} B{--} C{--} D{--}
# A cierra X
B: [RECEIVE]-> A{--} B{RECEIVE(OK,A)|TICK} C{--} D{--}
B: [RECEIVE]-> A{--} B{RECEIVE(OK,C)|TICK} C{--} D{--}
B: [RECEIVE]-> A{--} B{RECEIVE(OK,D)|TICK|MUTEX(Y)} C{--} D{--}
# B cierra Y
A: [EVENT]-> A{TICK} B{--} C{--} D{--}
B: [EVENT]-> A{--} B{TICK} C{--} D{--}
# Ahora liberamos X E Y
A: [UNLOCK X]-> A{--} B{--} C{--} D{--}
B: [UNLOCK Y]-> A{--} B{--} C{--} D{--}
# ##############################
# REPETIMOS LO MISMO CON C Y D #
# ##############################
C: [LOCK Y]-> A{--} B{--} C{TICK|SEND(LOCK,A)|SEND(LOCK,B)|SEND(LOCK,D)} D{--}
D: [LOCK X]-> A{--} B{--} C{--} D{TICK|SEND(LOCK,A)|SEND(LOCK,B)|SEND(LOCK,C)}
# Todos reciben las peticiones
```

```
C: [RECEIVE]-> A{--} B{--} C{RECEIVE(LOCK,D)|TICK|TICK|SEND(OK,D)} D{--}
B: [RECEIVE]-> A{--} B{RECEIVE(LOCK,C)|TICK|TICK|SEND(OK,C)} C{--} D{--}
A: [RECEIVE]-> A{RECEIVE(LOCK,C)|TICK|TICK|SEND(OK,C)} B{--} C{--} D{--}
D: [RECEIVE]-> A{--} B{--} C{--} D{RECEIVE(LOCK,C)|TICK|TICK|SEND(OK,C)}
B: [RECEIVE]-> A{--} B{RECEIVE(LOCK,D)|TICK|TICK|SEND(OK,D)} C{--} D{--}
A: [RECEIVE]-> A{RECEIVE(LOCK,D)|TICK|TICK|SEND(OK,D)} B{--} C{--} D{--}
# Tanto C como D deben recibir los 3 OKs
D: [RECEIVE]-> A{--} B{--} C{--} D{RECEIVE(OK,C)|TICK}
D: [RECEIVE]-> A{--} B{--} C{--} D{RECEIVE(OK,B)|TICK}
D: [RECEIVE]-> A{--} B{--} C{--} D{RECEIVE(OK,A)|TICK|MUTEX(X)}
# D cierra X
C: [RECEIVE]-> A{--} B{--} C{RECEIVE(OK,B)|TICK} D{--}
C: [RECEIVE]-> A{--} B{--} C{RECEIVE(OK,A)|TICK} D{--}
C: [RECEIVE]-> A{--} B{--} C{RECEIVE(OK,D)|TICK|MUTEX(Y)} D{--}
# C cierra Y
D: [EVENT]-> A{--} B{--} C{--} D{TICK}
C: [EVENT]-> A{--} B{--} C{TICK} D{--}
# Ahora liberamos X E Y
D: [UNLOCK X]-> A{--} B{--} C{--} D{--}
C: [UNLOCK Y]-> A{--} B{--} C{--} D{--}
# Terminando ?‘Valores de LCs?
A: [GETCLOCK]-> A{LC[12,5,6,6]} B{--} C{--} D{--}
B: [GETCLOCK]-> A{--} B{LC[4,13,6,6]} C{--} D{--}
C: [GETCLOCK]-> A{--} B{--} C{LC[10,11,12,8]} D{--}
D: [GETCLOCK]-> A{--} B{--} C{--} D{LC[12,13,8,12]}
FINISH[14906]
FINISH[14907]
FINISH[14908]
FINISH[14909]
```

4. Referencias

Documentación clase (Sincronización) .
Ejemplos de sockets de la página de la asignatura.

Registro de Revisiones

Este documento ha sido publicado como informe docente número DMUTEX'12 por el *Departamento de Arquitectura y Tecnología de Sistemas Informáticos* de la *Facultad de Informática* de la *Universidad Politécnica de Madrid* como parte de la asignatura *Sistemas Distribuidos*. El histórico de revisiones realizadas a este documento son:

- Revisión 4.9 (10/12/2011). Versión completa año 2012

- Versión inicial (06/05/2002).