TSR – 12 diciembre 2016. EJERCICIO 4

Se pide implementar en NodeJS y ZeroMQ el algoritmo de exclusión mutua con una sola sección crítica y servidor central atendiendo a las siguientes condiciones:

1) Se desarrollarán dos programas: por una parte el correspondiente al *servidor central* y por otra el correspondiente a los *procesos* que actúan sobre la sección crítica.

2) Todos los procesos que intervienen son anónimos y no tienen ninguna marca adicional que los permita distinguir entre sí.

3) El número de procesos puede, dentro de unos límites razonables, variar dinámicamente.

4) Ninguno de los mensajes intercambiados en el protocolo contiene información relevante (esto es, su contenido puede ser una cadena vacía o un simple espacio). Actúan únicamente como meros señalizadores.

5) Se utilizará un protocolo de gestión de la sección crítica que se ajuste al visto en el seminario 2; es decir:
   - Para solicitar el permiso, un proceso envía un mensaje al servidor y espera su respuesta concediéndole el permiso.
     - Si ningún proceso tiene el permiso el servidor le responde inmediatamente.
     - Si el permiso lo tiene otro proceso, el servidor no responde y encola la solicitud.
   - Cuando un proceso sale de la SC envía un mensaje al servidor devolviéndole el permiso. Si la cola de solicitudes no está vacía, el servidor selecciona la solicitud más antigua, la elimina de la cola y envía un mensaje otorgando el permiso al proceso correspondiente.

6) Para unificar las posibles soluciones, los procesos deben ajustarse a las siguientes reglas:
   - Tendrán dos variables lógicas denominadas `solicitarSC` (inicializada a **true**) y `abandonarSC` (inicializada a **false**).
   - Periódicamente, cada `1.5 seg`, se examinará el valor de la variable `solicitarSC`. Si está a **true** se solicitará al servidor central el permiso para acceder a la sección crítica.
   - Periódicamente, cada `2.0 seg`, se examinará el valor de la variable `abandonarSC`. Si está a **true** se notificará al servidor central el abandono de la sección crítica.
   - La computación llevada a cabo dentro de la sección crítica, en esta prueba, está definida por la función `seccionCritica` cuya única responsabilidad será asignar el valor **true** a `abandonarSC` tras haber transcurrido un segundo desde su invocación.

Solución (EJERCICIO 4)

Una posible solución sería la siguiente:

```
// Client.js
const zmq = require('zmq');

var solicitarSC = true;
var abandonarSC = false;
var request = zmq.socket('req');
var release = zmq.socket('push');

request.connect('tcp://127.0.0.1:8000');
release.connect('tcp://127.0.0.1:8001');

function seccionCritica() {
  console.log('Starting CS %d...', process.pid);
  setTimeout( function() {
    console.log('Releasing CS %d...',
      process.pid);
    abandonarSC = true;
  }, 1000);
}

function checkRequest() {
  if (solicitarSC) {
    solicitarSC = false;
    request.send('');
  }
}

request.on('message', seccionCritica );

setInterval( checkRequest, 1500 );

function checkRelease() {
  if (abandonarSC) {
    abandonarSC = false;
    release.send('');
    solicitarSC = true;
  }
}

setInterval( checkRelease, 2000 );
```

```
// Server.js
const zmq=require('zmq');

var request = zmq.socket('router');
var release = zmq.socket('pull');
request.bindSync('tcp://127.0.0.1:8000');
release.bindSync('tcp://127.0.0.1:8001');
var inCS = false;
var pending = [];

request.on('message', function(sender,
delimiter, msg) {
  if (!inCS) {
    request.send([sender,'', '']);
    console.log('CS assigned to a process.');
    inCS=true;
  } else {
    console.log('Pending request received
from a process...');
    pending.push(sender);
  }
});

release.on('message', function(m) {
  console.log('Released CS!');
  if (pending.length>0) {
    request.send([pending.shift(),'','']);
    console.log('CS assigned to a pending
process.');
  } else inCS = false;
});
```

En esta solución se han utilizado dos *sockets* en el programa servidor: uno para atender las solicitudes de entrada a la sección crítica y otro para atender las liberaciones de permiso. Como las solicitudes deben ser respondidas, se ha utilizado un *socket* bidireccional (**request,** de tipo ROUTER) en ese caso. Por su parte, las liberaciones no necesitan respuesta, por lo que se emplea un *socket* unidireccional (**release**, de tipo PULL).

El servidor utiliza una variable booleana (**inCS**) para saber si el permiso de acceso ya ha sido concedido o no.

Cuando llega una solicitud al *socket* **request** se comprueba si la sección crítica está libre o no. Si lo está, se contesta de inmediato y se pone a **true** la variable **inCS**. Si no lo está, se guarda la identidad del solicitante. Esa identidad se puede obtener implícitamente al utilizar un *socket* ROUTER para recibir las solicitudes. El primer segmento del mensaje recibido proporciona esa identidad implícita. Insertamos esa identidad en una cola **pending**, implantada mediante un vector.

Otra forma más sencilla de gestionar las solicitudes consiste en emplear un *socket* REP. En ese caso, la solución funcionará aun sin mantener las identidades de los procesos. Eso se debe a que el socket REP tiene un comportamiento "sincrónico". La primera solicitud recibe respuesta de inmediato. Eso hace que su proceso solicitante sea el "propietario" de la sección crítica. Con ello, una segunda solicitud podrá llegar y ser atendida, pero encontrará la SC ya ocupada y no tendrá que recibir ninguna respuesta todavía.

En este segundo caso, el código del servidor habría podido ser el siguiente:

```
// ServerRep.js
const zmq=require('zmq');

var request = zmq.socket('rep');
var release = zmq.socket('pull');
request.bindSync('tcp://127.0.0.1:8000');
release.bindSync('tcp://127.0.0.1:8001');
var inCS = false;
var pending = 0;

request.on('message', function(msg) {
    if (!inCS) {
      request.send('');
      console.log('CS assigned to a process.');
      inCS=true;
    } else {
      console.log('Pending request received from a process...');
      pending++;
    }
});

release.on('message', function(m) {
    console.log('Released CS!');
    if (pending >0) {
      request.send('');
      pending--;
      console.log('CS assigned to a pending process.');
    } else inCS = false;
});
```

Cuando llega un mensaje de liberación al *socket* **release**, deberá comprobarse si había alguna solicitud pendiente de contestación. Si la hay, debe enviarse una respuesta a la primera de ellas. En otro caso, bastará con dejar a valor **false** la variable **inCS**. De esa forma se expresa que la sección crítica está libre en ese momento.

Por lo que respecta al código del cliente, bastará con programar dos actividades periódicas mediante sus respectivos "setInterval()". En la primera actividad se comprueba el estado de la variable **solicitarSC**. Si está a **true** se tendrá que dejar con valor **false** y se enviará un mensaje vacío de solicitud al servidor.

En la segunda actividad se comprueba el estado de la variable **abandonarSC**. Si está a **true**, se fijará la variable **solicitarSC** a valor **true** y la variable **abandonarSC** a valor **false**, enviando también un mensaje de liberación al servidor.

Para enviar los mensajes de solicitud y liberación se emplearán dos sockets. En este ejemplo ha bastado con usar un REQ para las solicitudes y un PUSH para las liberaciones, pues eran los necesarios para implantar la bidireccionalidad en el canal de solicitudes y la unidireccionalidad en el de liberaciones. El tipo a emplear dependerá de los tipos seleccionados en el servidor.

Por último, para implantar la función **SeccionCritica**, basta con utilizar una sentencia setTimeout() que programe la asignación del valor **true** sobre la variable **abandonarSC** al cabo de 1000 milisegundos.

TSR – 12 diciembre 2016. EJERCICIO 5

Consideramos los siguientes programas

proc.js

```
var zmq= require('zmq')
var local = {x:0, y:0, z:0}
var port  = {x:9997, y:9998, z:9999}
var ws    = zmq.socket('push')
ws.connect('tcp://127.0.0.1:8888')
var rs    = zmq.socket('sub')
rs.subscribe("")
for (var i in port) rs.connect('tcp://127.0.0.1:'+port[i])
var id    = process.argv[2]

function W(name,value) {
  console.log("W"+id+"("+name+")"+value)
}
function R(name) {
  console.log("R"+id+"("+name+")"+local[name])
}

var n=0, names=["x","y","z"]
function writeValue() {
  n++; ws.send([names[n%names.length],(10*id)+n,id])
}

rs.on('message', function(name,value,writer) {
  local[name] = value
  if (writer == id) W(name,value); else R(name)
})

function work() { setInterval(writeValue,10) }
setTimeout(work,2000); setTimeout(process.exit,2500)
```

seq.js

```
const zmq= require('zmq')
var port = {x:9997, y:9998, z:9999}
var s    = {}

var pull = zmq.socket("'pull")
pull.bindSync('tcp://*:8888')

for (var i in port) {
    s[i]=zmq.socket('pub')
    s[i].bindSync('tcp://*:'+port[i])
}

pull.on('message',
  function(name,value,writer) {
    s[name].send([name,value,writer])
})
```

Ejecutamos desde el terminal la orden: **`node seq & node proc 1 & node proc 2 & node proc 3 &`**

Denominamos procesos a las instancias de ***proc*** y secuenciador a la única instancia de ***seq***. Procesos y secuenciador implementan una memoria compartida con variables ***x***,***y***,***z***. Los procesos trasmiten sus escrituras al secuenciador, y reciben desde el secuenciador las escrituras de los restantes procesos.

La ejecución genera la siguiente **traza**: `... R1(y)21 W1(y)11 W1(z)12 W2(y)21 R3(y)21 R2(z)12 R2(y)11 R3(y)11 R3(z)12 R1(z)22 W2(z)22 R3(z)22 W1(x)13 R3(x)13 R2(x)13 ...`

Analiza la **traza**, y en base a la misma responde **de forma razonada** a las siguientes cuestiones

1) (1.25 ptos) ¿soporta el modelo de consistencia **secuencial**?
2) (1.25 ptos) ¿soporta el modelo de consistencia **fifo**?
3) (1.25 ptos) ¿soporta el modelo de consistencia **caché**?

Analiza el **código,** y en base al mismo responde **de forma razonada** a las siguientes cuestiones

4) (1.75 ptos) Indica los sockets de cada componente y sus tipos, así como las conexiones establecidas entre los distintos sockets
5) (1.50 ptos) ¿soporta el modelo de consistencia **secuencial**?
6) (1.50 ptos) ¿soporta el modelo de consistencia **fifo**?
7) (1.50 ptos) ¿soporta el modelo de consistencia **caché**?

Solución (EJERCICIO 5)

Según se explicó en el Seminario 5, una operación `R(x)v` representa la recepción de un valor "v" escrito sobre la variable "x", donde ese evento de escritura ocurrió en otro proceso. Por su parte, la operación `W(x)v` representa la finalización de una escritura local, según el modelo de consistencia implantado.

Justifiquemos las respuestas en cada uno de los apartados de este ejercicio.

En la primera parte se pregunta sobre la secuencia de la **traza**:

1. Para que se respete el modelo **secuencial** todos los procesos deben estar de acuerdo en una única secuencia de valores en los que pueden aparecer todas las variables de la memoria compartida. Además, esa secuencia debe respetar también el orden de escritura de cada uno de los procesos que intervengan.

   Este modelo no llega a respetarse en esta ejecución. En P1 y P3 la secuencia observada ha sido: y=21, y=11, z=12, z=22, x=13, pero en P2 la secuencia observada ha sido otra: y=21, z=12, y=11, z=22, x=13. Como hemos podido encontrar dos secuencias distintas, estos tres procesos **no** respetan el modelo secuencial.

2. Para que se respete la consistencia **FIFO**, las escrituras realizadas por cada uno de los procesos deben ser observadas por los demás en el orden en que fueron escritas. En ese orden deben considerarse todas las variables.

   En esta traza hemos tenido cinco escrituras. Tres han sido realizadas por P1 (y=11, z=12, x=13) y dos por P2 (y=21, z=22). Atendiendo a las escrituras de P1, para que se respete el orden FIFO todos los procesos deben ver el valor 11 antes que el 12 y este último antes que el 13 (independientemente de las variables utilizadas). P2 no respeta esa secuencia de escritura realizada por P1, pues recibe el valor 12 antes que el 11. Por tanto, el modelo FIFO **no** está siendo respetado en esta ejecución.

   - Ése también sería un argumento válido para decir que no podía cumplirse el modelo secuencial, pues toda ejecución secuencial es FIFO (y también causal, procesador y caché).

3. El modelo **caché** exige que todos los procesos lleguen a un acuerdo sobre el orden en que deben ser observadas las escrituras sobre cada variable, considerando cada variable por separado. Es decir, hay libertad para intercalar las secuencias (comunes) de cada variable como cada lector prefiera.

   En esta traza se escribe dos veces sobre la variable "y" y dos más sobre la variable "z". Solo se escribe una vez en la variable "**x**". Por tanto, el único valor escrito sobre "x" no puede romper las condiciones impuestas por el modelo caché.

   Respecto a "**y**", P1 lee primero el valor 21 y después escribe el valor 11. Por tanto, los demás procesos deben seguir el mismo orden (primero 21 y después 11) para cumplir con el modelo caché. Tanto P2 como P3 respetan esa condición.

   Respecto a "**z**", P1 escribe primero el valor 12 y después lee el valor 22. Tanto P2 como P3 también observan esos dos valores en ese mismo orden. Por tanto, el orden de observación de los valores de las tres variables (por separado) es idéntico en los tres procesos y eso permite afirmar que el modelo caché se ha respetado.

En la segunda parte se ha de contestar a partir del **código**:

4. El secuenciador utiliza varios *sockets*:
    a. Necesita un *socket* **PULL** para recibir los mensajes de los procesos que comparten la memoria.
    b. Además de ese PULL utiliza **tantos *sockets* PUB como variables** haya en la memoria compartida. En este ejemplo tenemos tres variables ("x","y" y "z") por lo que se utilizan tres sockets PUB.

   Para cada uno de estos *sockets* se realiza una operación "bindSync()". De esta manera los procesos restantes deben realizar un "connect()" y con ello podrá haber tantos procesos lectores/escritores como se desee.
   Esos otros procesos utilizan, **cada uno de ellos, un *socket* PUSH** para comunicarle al secuenciador las escrituras que pretenden realizar (y que él ordenará) y **otro *socket* SUB** con el que se suscribe a todos los *sockets* PUB del secuenciador.
   De esta manera, los procesos que comparten la memoria tienen un canal PUSH-PULL unidireccional para hacer llegar sus escrituras al secuenciador y tres canales PUB-SUB (también unidireccionales) mediante los que reciben las difusiones del secuenciador. Como el secuenciador tiene tres *sockets* PUB independientes, cada uno de ellos está garantizando la misma secuencia para cada variable, pero pueden llegar a intercalar esas secuencias de distinta manera en cada receptor. Esto es, cada proceso observará la misma secuencia para cada variable considerada de manera aislada, pero no tienen por qué obtener la misma secuencia global (cuando se consideren todas las variables existentes conjuntamente).

5. El modelo **secuencial no** está siendo soportado por estos dos programas. Para que se soportase, debería utilizarse un único canal PUB/SUB para difundir las escrituras de todas las variables conjuntamente. Ahora tenemos un canal PUSH/PULL para que cada proceso, cuando escriba, haga llegar ese valor escrito al secuenciador. Como el canal PUSH/PULL utiliza TCP como transporte, se está garantizando el orden FIFO en esas propagaciones.
   Pero después se emplea un canal PUB/SUB diferente para difundir a todos los procesos lo que se ha escrito en cada variable. Esas propagaciones también son FIFO, pero al utilizar un canal distinto para cada variable, nada garantiza que todos los receptores observen la misma secuencia global, donde se siga un mismo orden para todas las variables. Por ello, el modelo secuencial no está soportado.

6. Los cuatro canales empleados, al utilizar TCP como transporte, mantienen el orden de envío a la hora de realizar las entregas en los procesos receptores. Por tanto, desde el escritor al secuenciador se pasan las acciones de escritura en orden FIFO y desde el secuenciador a los lectores también se estará respetando ese mismo orden, pero considerando cada variable por separado. Para que se respetase el modelo de consistencia FIFO no interesaría que el secuenciador propague cada variable por un canal de difusión distinto. Por ello, finalmente, los procesos lectores no siempre recibirán los valores escritos por un mismo proceso en el orden de escritura. Cuando esas escrituras afecten a variables distintas, la consistencia **FIFO no** siempre se respetará.

7. Esta versión de los programas garantiza la consistencia **caché**. ¿Por qué? Pues porque el uso de un canal PUB/SUB **para cada variable** sobre un transporte TCP nos garantiza que lo difundido **para cada variable** sea recibido en orden de difusión en los receptores. Con ello todos los procesos observan la misma secuencia en cada variable.