# TSR – 12 desembre 2016. EXERCICI 4

Es demana implementar en NodeJS i ZeroMQ l'algorisme d'exclusió mútua amb una sola secció crítica i servidor central atenent a les següents condicions:

1) S'escriuran dos programes: d'una banda el corresponent al *servidor central* i per una altra el corresponent als *processos* que actuen sobre la secció crítica.

2) Tots els processos que intervenen són anònims i no tenen cap marca addicional que els permeta distingir entre si.

3) El nombre de processos pot, dins d'uns límits raonables, variar dinàmicament.

4) Cap dels missatges intercanviats en el protocol conté informació rellevant (és a dir, el seu contingut pot ser una cadena buida o un simple espai). Actuen únicament com a senyalitzadors.

5) S'utilitzarà un protocol de gestió de la secció crítica que s'ajuste al vist en el seminari 2; és a dir:
   - Per a sol·licitar el permís, un procés envia un missatge al servidor i espera la seua resposta concedint-li el permís.
     - Si cap procés té el permís el servidor li respon immediatament.
     - Si el permís el té un altre procés, el servidor no respon i encua la sol·licitud.
   - Quan un procés ix de la SC envia un missatge al servidor retornant-li el permís. Si la cua de sol·licituds no està buida, el servidor selecciona la sol·licitud més antiga, l'elimina de la cua i envia un missatge atorgant el permís al procés corresponent.

6) Per a unificar les possibles solucions, els processos han d'ajustar-se a les següents regles:
   - Tindran dues variables lògiques denominades `solicitarSC` (inicialitzada a **true**) i `abandonarSC` (inicialitzada a **false**).
   - Periòdicament, cada `1.5 seg`, s'examinarà el valor de la variable `solicitarSC`. Si està **a true** se sol·licitarà al servidor central el permís per a accedir a la secció crítica.
   - Periòdicament, cada `2.0 seg`, s'examinarà el valor de la variable `abandonarSC`. Si està **a true** es notificarà al servidor central l'abandó de la secció crítica.
   - La computació duta a terme dins de la secció crítica, en aquesta prova, està definida per la funció `seccionCritica` l'única responsabilitat de la qual serà assignar el valor **true** a `abandonarSC` després d'haver transcorregut un segon des de la seua invocació.

# TSR – 12 desembre 2016. EXERCICI 5

Considerem els següents programes

**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])
})
```

Executem des del terminal l'ordre: **`node seq & node proc 1 & node proc 2 & node proc 3 &`**

Denominem processos a les instàncies de ***proc*** i seqüenciador a l'única instància de ***seq***. Processos i seqüenciador implementen una memòria compartida amb variables ***x***,***y***,***z***. Els processos transmeten les seues escriptures al seqüenciador, i reben des del seqüenciador les escriptures dels restants processos.

L'execució genera la següent **traça**: `... 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 ...`

Analitza la **traça**, i sobre la base de la mateixa respon **de forma raonada** a les següents qüestions

1) (1.25 punts) Suporta el model de consistència **seqüencial**?
2) (1.25 punts) Suporta el model de consistència **fifo**?
3) (1.25 punts) Suporta el model de consistència "**cache**"?

Analitza el **codi,** i sobre la base del mateix respon **de forma raonada** a les següents qüestions

4) (1.75 punts) Indica els sockets de cada component i els seus tipus, així com les connexions establides entre els diferents sockets
5) (1.50 punts) Suporta el model de consistència **seqüencial**?
6) (1.50 punts) Suporta el model de consistència **fifo**?
7) (1.50 punts) Suporta el model de consistència "**cache**"?

## SOLUCIÓ DE L'EXERCICI 4

Una possible solució seria la següent:

```
// 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 aquesta solució s'han utilitzat dos *sockets* en el programa servidor: un per a atendre les sol·licituds d'entrada a la secció crítica i un altre per a atendre els alliberaments de permís. Com les sol·licituds han de ser respostes, s'ha utilitzat un *socket* bidireccional (**request,** de tipus ROUTER) en aquest cas. Per la seua banda, els alliberaments no necessiten resposta, per la qual cosa s'empra un *socket* unidireccional (**release**, de tipus PULL).

El servidor utilitza una variable booleana (**inCS**) per a saber si el permís d'accés ja ha sigut concedit o no.

Quan arriba una sol·licitud al *socket* **request** es comprova si la secció crítica està lliure o no. Si ho està, es contesta immediatament i es posa a **true** la variable **inCS**. Si no ho està, es guarda la identitat del sol·licitant. La identitat es

pot obtenir implícitament en utilitzar un *socket* ROUTER per a rebre les sol·licituds. El primer segment del missatge rebut proporciona la identitat implícita. Inserim la identitat en una cua **pending**, implantada mitjançant un vector.

Una altra forma més senzilla de gestionar les sol·licituds consisteix a emprar un *socket* REP. En aquest cas, la solució funcionarà sense mantenir les identitats dels processos. Això es deu al fet que el socket REP té un comportament "sincrònic". Quan arriba la primera petició es pot contestar immediatament. Així, el procés sol·licitant es converteix en el "propietari" de la SC. Aleshores, el socket REP queda "lliure". Amb això admetrà una segona petició, que processarà, però no podrà contestar encara per a seguir correctament l'algorisme (quedarà en un estat "pendent" i serà contestada quan el propietari actual allibere la SC). Si es reben altres peticions d'altres processos, quedaran totes elles en la cua d'entrada del socket REP. Aleshores, no hi ha forma de creuar les respostes entre dos sol·licitants diferents. En aquest segon cas, el codi del servidor hauria pogut ser el següent:

```
// 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;
});
```

Quan arriba un missatge d'alliberament al *socket* **release**, haurà de comprovar-se si hi havia alguna sol·licitud pendent de contestació. Si n'hi ha alguna, ha d'enviar-se una resposta a la primera d'elles. En un altre cas, només caldrà deixar a **false** la variable **inCS**. D'aquesta forma s'expressa que la secció crítica està lliure en aquest moment.

Pel que fa al codi del client, només cal programar dues activitats periòdiques mitjançant els seus respectius "setInterval()". En la primera activitat es comprova l'estat de la variable **solicitarSC**. Si està a **true** s'haurà de deixar amb valor **false** i s'enviarà un missatge buit de sol·licitud al servidor.

En la segona activitat es comprova l'estat de la variable **abandonarSC**. Si està a **true**, es fixarà la variable **solicitarSC** a valor **true** i la variable **abandonarSC** a valor **false**, enviant també un missatge d'alliberament al servidor.

Per a enviar els missatges de sol·licitud i alliberament s'empraran dos sockets. En aquest exemple ha bastat amb usar un REQ per a les sol·licituds i un PUSH per als alliberaments, doncs eren els necessaris per a implantar la

bidireccionalitat en el canal de sol·licituds i la unidireccionalitat en el d'alliberaments. El tipus a emprar dependrà dels tipus seleccionats en el servidor.

Finalment, per a implantar la funció **SeccionCritica**, n'hi ha prou amb utilitzar una sentència setTimeout() que programe l'assignació del valor **true** sobre la variable **abandonarSC** al cap de 1000 mil·lisegons.

## SOLUCIÓ DE L'EXERCICI 5

Segons es va explicar en els enunciats del Seminari 5, una operació R(x)v representa la recepció d'un valor "v" escrit sobre la variable "x", on aquest esdeveniment d'escriptura va ocórrer en un altre procés. Per la seua banda, l'operació W(x)v representa la finalització d'una escriptura local, segons el model de consistència implantat.

Justifiquem les respostes en cadascun dels apartats d'aquest exercici.

1. Perquè es respecte el model seqüencial tots els processos han d'estar d'acord en una única seqüència de valors en els quals poden aparèixer totes les variables de la memòria compartida. A més, la seqüència ha de respectar també l'ordre d'escriptura de cadascun dels processos que intervinguen.
   Aquest model no arriba a respectar-se en aquesta execució. En P1 i P3 la seqüència observada ha sigut: y=21, y=11, z=12, z=22, x=13, però en P2 la seqüència observada ha sigut una altra: y=21, **z=12, y=11**, z=22, x=13. Com hem pogut trobar dues seqüències diferents, aquests tres processos no respecten el model seqüencial.
2. Perquè es respecte la consistència FIFO, les escriptures realitzades per cadascun dels processos han de ser observades pels altres en l'ordre en què van ser escrites. En aquest ordre han de considerar-se totes les variables. En aquesta traça hem tingut cinc escriptures. Tres han sigut realitzades per P1 (y=11, z=12, x=13) i dues per P2 (y=21, z=22). Atenent a les escriptures de P1, per a que es respecte l'ordre FIFO tots els processos han de veure el valor 11 abans que el 12 i aquest últim abans que el 13 (independentment de les variables utilitzades). P2 no respecta aquesta seqüència d'escriptura realitzada per P1, perquè rep el valor 12 abans que l'11. Per tant, el model FIFO no està sent respectat en aquesta execució. Aquest també seria un argument vàlid per a dir que no podia complir-se el model seqüencial ja que tota execució seqüencial és FIFO (i també causal, processador i "cache").
3. El model "cache" exigeix que tots els processos arriben a un acord sobre l'ordre en què han de ser observades les escriptures sobre cada variable, considerant cada variable per separat. És a dir, hi ha llibertat per a intercalar les seqüències de cada variable com cada lector preferisca.
   En aquesta traça s'escriu dues vegades sobre la variable "y" i dues més sobre la variable "z". Solament s'escriu una vegada en la variable "x". Per tant, l'únic valor escrit sobre "x" no pot trencar les condicions imposades pel model "cache".
   Respecte a "y", P1 llig primer el valor 21 i després escriu el valor 11. Per tant, els altres processos han de seguir el mateix ordre (primer 21 i després 11) per a complir amb el model "cache". Tant P2 com a P3 respecten aquesta condició.
   Respecte a "z", P1 escriu primer el valor 12 i després llig el valor 22. Tant P2 com P3 també observen aquests dos valors en aquest mateix ordre. Per tant, l'ordre d'observació dels valors de les tres variables (per separat) és idèntic en els tres processos i això permet afirmar que el model "cache" s'ha respectat.
4. El seqüenciador utilitza múltiples *sockets*:
   a. Necessita un *socket* PULL per a rebre els missatges dels processos que comparteixen la memòria.
   b. A més d'aquest PULL utilitza tants *sockets* PUB com a variables hi haja en la memòria compartida. En aquest exemple tenim tres variables ("x","y" i "z") i per això s'utilitzen tres sockets PUB.

   Cadascun d'aquests *sockets* realitza una operació "bindSync()". D'aquesta manera els processos restants han de realitzar un "connect()" i amb això podrà haver-hi tants processos lectors/escriptors com es desitge.
   Aquests altres processos utilitzen, cadascun d'ells, un *socket* PUSH per a comunicar-li al seqüenciador les escriptures que pretenen realitzar (i que ell ordenarà) i un altre *socket* SUB amb el qual se subscriuen a *tots els sockets* PUB del seqüenciador.
   D'aquesta manera, els processos que comparteixen la memòria tenen un canal PUSH-PULL unidireccional per a fer arribar les seues escriptures al seqüenciador i tres canals PUB-SUB (també unidireccionals) mitjançant els quals reben les difusions del seqüenciador. Com el seqüenciador té tres *sockets* PUB independents,

cadascun d'ells està garantint la mateixa seqüencia per a cada variable, però poden arribar a intercalar aquestes seqüències de diferent manera en cada receptor. És a dir, cada procés observarà la mateixa seqüencia per a cada variable considerada de manera aïllada, però no tenen per què obtenir la mateixa seqüència global (quan es consideren totes les variables existents conjuntament).

5. El model seqüencial no està sent suportat per aquests dos programes. Perquè se suportara, hauria d'utilitzar-se un únic canal PUB/SUB per a difondre les escriptures de totes les variables conjuntament. Ara tenim un canal PUSH/PULL per a què cada procés, quan escriga, faça arribar el valor escrit al seqüenciador. Com el canal PUSH/PULL utilitza TCP com a transport, s'està garantint l'ordre FIFO en les propagacions.
   Però després s'empra un canal PUB/SUB diferent per a difondre a tots els processos el que s'ha escrit en cada variable. Aquestes propagacions també són FIFO, però en utilitzar un canal diferent per a cada variable, no es garanteix que tots els receptors observen la mateixa seqüència global, on se seguisca un mateix ordre per a totes les variables. Per això, el model seqüencial no està suportat.
6. Els quatre canals emprats, en utilitzar TCP com a transport, mantenen l'ordre d'enviament a l'hora de realitzar els lliuraments en els processos receptors. Per tant, des de l'escriptor al seqüenciador es passen les accions d'escriptura en ordre FIFO i des del seqüenciador als lectors també s'estarà respectant aquest mateix ordre, però considerant cada variable per separat. Perquè es respectara el model de consistència FIFO no interessaria que el seqüenciador propague cada variable per un canal de difusió diferent. Per això, finalment, els processos lectors no sempre rebran els valors escrits per un mateix procés en l'ordre d'escriptura. Quan les escriptures afecten a variables diferents, la consistència FIFO no sempre es respectarà.
7. Aquesta versió dels programes garanteix la consistència “cache”. Per què? Perquè l'ús d'un canal PUB/SUB per a cada variable sobre un transport TCP ens garanteix que el difós per a cada variable siga rebut en ordre de difusió en els receptors. Amb això tots els processos observen la mateixa seqüència en cada variable.