TSR – 12th December 2016. EXERCISE 4

Please implement in NodeJS and ZeroMQ the central server mutual exclusion algorithm considering the following conditions:

1) Two programs are needed: one for the central server and another for critical section requestors.

2) All intervening processes are anonymous and they don't use any kind of tag allowing their identification.

3) The amount of running processes may vary dynamically.

4) Messages do not contain any relevant information (indeed, they must be either an empty string or a single blank).

5) These programs implement the mutual exclusion algorithm seen in Seminar 2; i.e.:
   - In order to request permission, a process sends a message to the server and waits for its response (granting that permission).
     - If no process is in the critical section, the server responds immediately.
     - If a process is in the critical section, the server doesn't reply and enqueues that request.
   - When a process completes its critical section, it sends a message to the server, releasing its permission. If the pending request queue isn't empty, the server chooses its oldest request, dequeuing it, and sends an answer message to that requestor.

6) The requesting processes should respect the following rules:
   - They should manage two logical variables: `solicitarSC` (*require critical section*, initially **true**) and `abandonarSC` (*release critical section*, initially **false**).
   - Every 1.5 seconds, the process checks the value of `solicitarSC`. If it is **true**, the process sends a message to the central server requesting its access to the critical section.
   - Every 2.0 seconds, the process checks the value of `abandonarSC`. If it is **true**, the process sends a message to the central server releasing its critical section access permission.
   - The execution of the critical section is modelled by a function named `seccionCritica` (*critical section*) whose single purpose is to assign a **true** value to `abandonarSC`. However, that assignment must be done one second after this function (i.e., `seccionCritica`) was called.

TSR – 12th December 2016. EXERCISE 5

Let us consider the following programs:

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])
})
```

Consider the following command line: **`node seq & node proc 1 & node proc 2 & node proc 3 &`**

Let us call "processes" the instances of ***proc*** and "sequencer" the single instance of ***seq***. Processes and sequencer implement a shared memory with variables ***x***,***y***,***z***. Processes forward their write actions to the sequencer and receive from the sequencer the writings of other processes.

This execution generates this **trace**: `... 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 ...`

Given that trace, please **justify** your answers to the following questions:

1) (1.25 points) Does this trace respect the **sequential** consistency model?
2) (1.25 points) Does this trace respect the **FIFO** consistency model?
3) (1.25 points) Does this trace respect the **cache** consistency model?

Regarding these programs, please justify (referring to their functions and relevant fragments of **code**) your answers to the following questions:

4) (1.75 points) Explain which sockets (i.e., which types of sockets) and which communication patterns have been used by these processes.
5) (1.50 points) Do these programs implement the **sequential** consistency model?
6) (1.50 points) Do these programs implement the **FIFO** consistency model?
7) (1.50 points) Do these programs implement the **cache** consistency model?

EXERCISE 4: SOLUTION

This problem admits multiple solutions. A first one could be:

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

Note that all the **console.log()** statements are optional (they were not mentioned in the problem wording). In the following paragraphs we will use the CS initials in order to refer to the "critical section".

This solution uses two sockets in the server process: one for the CS-entry requests and another for the CS-exit messages. CS-entry requests demanded a response. This means that they should be managed with a bidirectional socket (**request**, a ROUTER socket). On the other hand CS-exit messages do not require any answer. Therefore, they can be managed with a unidirectional socket (**release**, a PULL socket).

The server uses a Boolean variable (**inCS**) in order to know whether the CS is busy or not.

When a request message is received in the **request** socket, the server checks whether the CS is busy or not. If not, it assigns a **true** value to **inCS**. Otherwise, the identity of the requestor is saved. That identity (actually, a connection ID) is passed in the first segment of every incoming message, since we have used a ROUTER socket. That ID is pushed to the **pending** array, which is managed as a request queue.

Another variant for request management consists in using a REP socket. In this variant, no requestor ID is needed. Note that REP sockets have a "synchronous" behaviour. Because of this, the first received request is immediately replied, and the CS is assigned to that process. This frees the REP socket, and it may receive a second request. That second request finds the CS busy. As a result, its reply cannot be sent yet. All the subsequent ones are held in the incoming queue and are not delivered yet. REP sockets avoid any danger of reply interleaving. With this second strategy, the server program could have been similar to this:

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

When a CS-exit message is received in the **release** socket, the server should check whether there was any pending CS-entry request in the **pending** queue. If so, an answer should be sent to the first one (that is extracted from the queue using the shift() method). Otherwise, the server should set **inCS** to **false**. Thus, the server knows that the CS remains free from now on.

Regarding the client program, it is mainly concerned with the periodical management of two activities (programmed using "setInterval()"). The first activity consists in checking the value of **solicitarSC** (i.e., requestCS). If it is **true** we should set it to **false** and send an empty CS-entry request to the server.

The second activity consists in checking the value of **abandonarSC** (i.e., releaseCS). If it is **true**, we should set **solicitarSC** to **true** (this is optional, but recommendable) and **abandonarSC** to **false**, sending also an empty CS-exit message to the server.

In order to send these messages we need two sockets. In this proposal we have used a REQ socket for the CS-entry requests and a PUSH one for CS-exit messages, since they were the easiest options for implementing a bidirectional channel for requests and a unidirectional channel for release messages.The socket types to be used depend on the sockets that have been chosen at the server side.

Finally, for implementing **SeccionCritica**, we should use a setTimeout() statement whose programmed function should set **abandonarSC** to **true** in 1000 milliseconds.

EXERCISE 5: SOLUTION

According to the explanations given in Seminar 5, a Rp(x)v operation means the value "v" for variable "x" **has been received** by process "p". Also, Wp(x)v means that process "p" **has completed** a write operation on "x" according to the consistency model being assumed.

Let us justify the answers to each part of this exercise:

1. In order to respect the sequential consistency model all processes must agree on a unique sequence of values for all shared variables. Besides, that sequence should also respect the writing order of each participating process.
   This model is not respected in this execution. P1 and P3 have seen the following sequence: y=21, y=11, z=12, x=22, x=13, but the sequence observed by P2 is different: y=21, **z=12, y=11**, z=22, x=13. Due to this, the consistency model in that execution is not sequential.
2. In order to respect FIFO consistency, the writes done by each one of the participating processes should be seen by the remaining processes in writing order. That order includes all variables. This trace consists of five writes. Three have been written by P1 (y=11, z=12, x=13) and two by P2 (y=21, z=22). Considering P1 writes, all readers should see value 11 before value 12 and this latter before value 13. P2 doesn't observe those values in the intended order, since it has received value 12 before receiving value 11. Due to this, FIFO consistency is not respected in this execution. Since sequential consistency is stronger than FIFO, this same argument could have been used for justifying that sequential consistency wasn't respected.
3. Cache consistency requires that all processes see the same sequence of values on each variable, considering each variable separately from the others. Due to this, those per-variable sequences may be arbitrarily interleaved by each reader.
   This execution writes twice on "y" and twice on "z", but only once on "x". Due to this, the single write on "x" trivially respects the requirements of cache consistency.
   Regarding "y", P1 reads first value 21 and later writes value 11. Because of this, all remaining processes should see those values in that order (21 before 11). Both P2 and P3 respect that constraint.
   Regarding "z", P1 writes first value 12 and later reads value 22. Both P2 and P3 observe those values in the same order. As a result, all requirements of cache consistency are respected in this execution and we may assert that the cache consistency model is maintained in this exercise.
4. The sequencer process uses several sockets:
   a. It needs a PULL socket for receiving messages from the other processes.
   b. Besides that PULL socket, it also suses as many PUB sockets as shared variables exist in the memory. In this example, we have three variables ("x", "y" and "z"); so, three PUB sockets are used.

   Each one of these sockets uses the "bindSync()" operation. Thus, the processes that use the shared memory need to use a "connect()" operation. With this, this architecture may tolerate as many reader/writer processes as needed.
   Those other processes use, each one of them, a PUSH socket for sending to the sequencer their write actions and another SUB socket in order to subscribe to all sequencer PUB sockets.
   Thus, reader/writer processes have a PUSH-PULL undirectional channel for sending their write actions to the sequencer and three PUB-SUB (unidirectional) channels for receiving the sequencer multicasts. As the sequencer has three independent PUB sockets, each one of them is guaranteeing the same sequence of values for each variable, but those sequences may be arbitrarily interleaved by each receiver. As a result, each process observes the same sequence of values per variable, but not necessarily the same global (i.e., considering jointly all the shared variables) sequence.
5. The sequential model is not respected by these two programs. In order to be respected, they should use a unique PUB/SUB channel for jointly multicasting all written values on all variables. Now, we have a PUSH/PULL channel for writer-sequencer communication. That channel respects FIFO delivery. But later, a different PUB/SUB channel is used for each variable. Those multicast messages also respect a FIFO order in

each channel, but the values received from each channel may be arbitrarily interleaved by each receiver. Due to this, the sequential model is not ensured.

6. Since all four communication channels use a TCP transport, all of them follow a FIFO-ordered message delivery. In spite of this, since the write actions of a given process (when applied to different variables) may use different PUB/SUB channels in order to be delivered to the other processes, their reception may not follow a FIFO order. Because of this, FIFO consistency is not respected by these programs.
7. As it has been explained at the end of part 4, the usage of these four communication channels respects all the constraints imposed by the cache consistency model. Because of this, these two programs implement the cache consistency model.