TSR: Tema 5. Actividades

En este documento se proponen actividades relacionadas con los algoritmos necesarios para asegurar algunos tipos de consistencia en sistemas distribuidos. Se asumirá la siguiente notación para representar una operación en una ejecución distribuida:

<operación><proceso> **(** <variable> **)** <valor>

Donde <proceso> es un identificador de proceso, <operación> será "W" para las escrituras y "R" para las lecturas, <variable> es un identificador de variable y <valor> representa el valor escrito en (o leído de) la variable correspondiente.

La operación "R" no implica necesariamente que el proceso realice una lectura en ese momento sino que desde ese momento el valor que obtendrá en una lectura será el indicado en esa operación. Por tanto, R1(x)5 no indica que P1 esté leyendo el valor 5 sino que a partir de ese punto el protocolo de consistencia utilizado ha conseguido transmitir al proceso 1 que el nuevo valor para la variable "x" es un 5. Cualquier lectura realizada a partir de ese momento retornará el valor 5, hasta que P1 ejecute una nueva recepción de valor o "W1(x)".

Si un proceso realiza una acción de escritura, se entenderá que cualquier lectura local obtendría a partir de ese momento ese valor escrito. Ese valor se mantendrá hasta la siguiente operación "R" o "W".

Nuestro objetivo será lanzar varios procesos en un mismo ordenador, a ser posible desde una misma consola, para recoger toda la información que escribe este conjunto de procesos en pantalla en una sola secuencia. De esa manera resultará sencillo volcar toda la salida sobre un mismo fichero para analizarla con calma posteriormente.

Los enunciados de cada cuestión facilitan un pequeño fragmento de la salida obtenida. Con ello se podrá responder a las preguntas sin necesidad de ejecutar los programas.

Los ficheros mencionados en este documento se incluyen en un ZIP llamado implCons.zip que puede descargarse desde PoliformaT, en la carpeta correspondiente a este tema.

ACTIVIDAD 1

Se ha implantado cierto modelo de consistencia relajado utilizando este módulo NodeJS:

```
/*************************************************************************/
/* shared1.js                                                            */
/* Module that supports a non-strict consistency model.                  */
/*************************************************************************/

// 0MQ is used as the communication module.
var zmq = require('zmq');

// The publish/subscribe interaction pattern is used. A socket of each
// kind is needed.
var pb = zmq.socket('pub');
var sb = zmq.socket('sub');
// Global subscription.
sb.subscribe("");
// The "local" object emulates the shared memory. Each one of its
// attributes is a shared variable.
var local = {};
// In order to be easy to use, this module assumes that all processes
// are deployed onto the same computer. So, each one uses a different
// port. To this end, a common prefix is used. Each process uses its
// identifier as a suffix of such port number.
var prefixPort = "1000";

/*=======================================================================*/
/* PRIVATE FUNCTIONS                                                     */
/*=======================================================================*/

/*************************************************************************/
/* join(nodes)                                                           */
/* The local process uses this function to connect its "subscriber"      */
/* socket to the "publisher" sockets of the remaining processes.         */
/* Input args:                                                           */
/* - nodes: An array with the endpoints of all system processes.         */
/*************************************************************************/
function join(nodes) {
    // Iterate onto the "nodes" array, connecting the local subscriber
    // socket to the endpoint held in each array slot.
    nodes.forEach(function (ep) {
        sb.connect(ep);
    });
}

// The local process ID is initially set to -1.
var id = -1;

/*************************************************************************/
/* myID(nid)                                                             */
/* Sets the local process ID and binds its publisher socket to its       */
/* intended port.                                                        */
/* Input args:                                                           */
/* - nid: Process identifier in the range 0..9                           */
/*************************************************************************/
function myID(nid) {
    // Checks whether the function has already been called.
    if (id!=-1)
        // If so, return without doing anything else.
        return;
    // Save the process identifier. "id" is a global variable.
    id = parseInt(nid);
    // If the identifier has a wrong value, print an error message and
    // abort.
    if (isNaN(id) || id < 1 || id > 9) {
        console.log("ERROR: The node identifier should be a number " +
                    "in the range 1..9!!");
        process.exit();
    }
    // Bind the publisher socket to the appropriate port.
    pb.bindSync("tcp://127.0.0.1:"+prefixPort+id);
}

/*=======================================================================*/
/* PUBLIC FUNCTIONS                                                      */
```

```
 73: /*=========================================================================*/
 74:
 75: /*************************************************************************/
 76: /* init(id)                                                              */
 77: /* Initialises the "shared1" module, receiving the identifier of the     */
 78: /* invoker local process.                                                */
 79: /* Input args:                                                           */
 80: /* - id: Local process identifier.                                       */
 81: /*************************************************************************/
 82: exports.init = function(id) {
 83:     var i;
 84:     var nodes=[];
 85:     for (i=0; i<10; i++)
 86:        nodes.push("tcp://127.0.0.1:"+prefixPort+i);
 87:     join(nodes);
 88:     myID(id);
 89: }
 90:
 91: // Prefix of the port number to be used.
 92: exports.prefixPort = prefixPort;
 93:
 94: /*************************************************************************/
 95: /* W(name,value)                                                         */
 96: /* Writes the "value" value onto the "name" shared variable.             */
 97: /* Input args:                                                           */
 98: /* - name: String with the name of the shared variable being written.    */
 99: /* - value: Value to be written.                                         */
100: /*************************************************************************/
101: exports.W = function (name, value) {
102:     // Write a message on standard output describing this write action.
103:     // We do not use console.log() since it always writes a final newline
104:     // character.
105:     process.stdout.write("W" + id + "(" + name + ")" + value + ":" );
106:     // Propagate <name,value,pid> to all system processes.
107:     pb.send([name, value, id]);
108:     // Write also such value onto the local copy of the "name" variable.
109:     local[name] = value;
110: };
111:
112: /*************************************************************************/
113: /* R(name)                                                               */
114: /* Reads the current value of the "name" shared variable.                */
115: /* Input args:                                                           */
116: /* - name: String with the name of the shared variable being read.       */
117: /* Return value:                                                         */
118: /* - Current value of such variable.                                     */
119: /*************************************************************************/
120: exports.R = function (name) {
121:     // Write a message on standard output describing this read action.
122:     process.stdout.write("R" + id + "(" + name + ")" + local[name] + ":");
123:     // Return the read value.
124:     return local[name];
125: };
126:
127: /*************************************************************************/
128: /* 'message' event handler                                               */
129: /* The messages being broadcast through the pub/sub channel consist of   */
130: /* three segments: "name", "value" and "sender".                         */
131: /* We store the received value onto the local copy of the "name"         */
132: /* variable.                                                             */
133: /* In order to follow the trace, a "read" event is printed on screen.    */
134: /*************************************************************************/
135: sb.on('message', function (name, value, sender) {
136:     if (sender != id) {
137:         // Set the received value onto the local copy of the variable.
138:         local[name] = value;
139:         // Print a message on screen, emulating a local read.
140:         exports.R(name);
141:     }
142: });
```

Para probar el módulo “shared1.js” se ha escrito este programa:

```
 1: /*************************************************************************/
 2: /*                                                                       */
 3: /* proc1.js                                                              */
 4: /*                                                                       */
 5: /* This program emulates a process in a distributed system with a set    */
 6: /* of shared variables.                                                  */
 7: /*                                                                       */
 8: /* The process identifier should be passed as an argument from the       */
 9: /* command line.                                                         */
10: /*                                                                       */
11: /*************************************************************************/
12:
13: sh = require("./shared1");
14: var myID; // Process identifier.
15: var multipliers = [11,13,17,19,23,29,31,37,41,43,47,53,59];
16:
17: /*************************************************************************/
18: /* getID()                                                               */
19: /* Returns the first argument given in the command line.                 */
20: /* Prints an error message and aborts the program if none is given.      */
21: /*                                                                       */
22: /*************************************************************************/
23: function getID() {
24:     // Get the command-line arguments.
25:     args = process.argv;
26:     // Check whether at least an argument has been given.
27:     if (args.length < 3) {
28:         // If not, print an error message and exit.
29:         console.log("ERROR: Please enter the process identifier as the " +
30:                     "first argument in the command line!");
31:         process.exit();
32:     }
33:     // Otherwise, return it as the process ID.
34:     return parseInt(args[2]);
35: }
36:
37: /*************************************************************************/
38: /* writeValue()                                                          */
39: /* Write a value derived from the process identifier onto "x" (the       */
40: /* shared variable).                                                     */
41: /*************************************************************************/
42: var counter=0;
43: function writeValue() {
44:     // Write the process ID onto variable "x".
45:     sh.W("x",myID*multipliers[counter++ % multipliers.length]);
46: }
47:
48: /*************************************************************************/
49: /* myActivity()                                                          */
50: /* Write a value every 10 ms.                                            */
51: /*************************************************************************/
52: function myActivity() {
53:     setInterval(writeValue, 10);
54: }
55:
56: // Obtains the process ID.
57: myID = getID();
58: // Initialises the shared1 module.
59: sh.init(myID);
60: sh.W("y", myID); // Unimportant write.
61: // Let the process start in two seconds.
62: setTimeout(myActivity,2000);
63: // Terminate the process after some time.
64: setTimeout(process.exit,3000);
```

Se han iniciado tres procesos que ejecutan ese código mediante la orden:

```
node proc1 1 & node proc1 2 & node proc1 3 &
```

La salida obtenida en esa ejecución ha sido la siguiente:

```
1: W2(y)2:W1(y)1:W3(y)3:W1(x)11:W2(x)22:R3(x)11:R3(x)22:R1(x)22:W3(x)33:R2(x)11:R2(x
2: )33:R1(x)33:W2(x)26:W1(x)13:R3(x)13:R2(x)13:R3(x)26:R1(x)26:W3(x)39:R1(x)39:R2(x)
3: 39:W1(x)17:R3(x)17:R2(x)17:W2(x)34:R3(x)34:R1(x)34:W3(x)51:R2(x)51:R1(x)51:W2(x)3
4: 8:W1(x)19:R1(x)38:R2(x)19:R3(x)38:R3(x)19:W3(x)57:R1(x)57:R2(x)57:W1(x)23:W2(x)46
5: :R3(x)23:R2(x)23:R3(x)46:R1(x)46:W3(x)69:R2(x)69:R1(x)69:W2(x)58:W1(x)29:R3(x)58:
```

```
 6: R1(x)58:R2(x)29:R3(x)29:W3(x)87:R2(x)87:R1(x)87:W2(x)62:W1(x)31:R2(x)31:R1(x)62:R
 7: 3(x)62:R3(x)31:W3(x)93:R2(x)93:R1(x)93:W2(x)74:W1(x)37:R1(x)74:R2(x)37:R3(x)74:R3
 8: (x)37:W3(x)111:R2(x)111:R1(x)111:W2(x)82:R3(x)82:R1(x)82:W1(x)41:R2(x)41:R3(x)41:
 9: W3(x)123:R2(x)123:R1(x)123:W2(x)86:R3(x)86:R1(x)86:W1(x)43:R3(x)43:R2(x)43:W3(x)1
10: 29:R2(x)129:R1(x)129:W2(x)94:R3(x)94:R1(x)94:W1(x)47:R3(x)47:R2(x)47:W3(x)141:R1(
11: x)141:R2(x)141:W1(x)53:W2(x)106:R1(x)106:R3(x)53:R3(x)106:R2(x)53:W3(x)159:R2(x)1
12: 59:R1(x)159:W2(x)118:W1(x)59:R1(x)118:R3(x)118:R3(x)59:R2(x)59:...
```

Responda y justifique las siguientes cuestiones relacionadas con este ejemplo:

1. En base a los dos fragmentos de la traza de ejecución que se han resaltado en color amarillo en la tabla anterior, justifique si el modelo de consistencia que se está proporcionando en este ejemplo es alguno de los siguientes:
    a. Estricto.
    b. Secuencial.
    c. Caché.
2. Revise con cuidado el código del módulo “shared1.js” y justifique qué modelo o modelos de consistencia puede implantar ese módulo. Indique en qué líneas de código se basa para realizar su justificación.

ACTIVIDAD 2

Tomando como base el módulo “shared1.js” de la actividad anterior, se ha desarrollado esta variante:

```
 1: /*************************************************************************/
 2: /* shared2.js                                                            */
 3: /* Module that supports a non-strict consistency model.                  */
 4: /*************************************************************************/
 5:
 6: // 0MQ is used as the communication module.
 7: var zmq = require('zmq');
 8:
 9: // Two interaction patterns are used. In the first step, each process uses
10: // a push/pull channel to forward its message to the sequencer. Later,
11: // the sequencer uses a pub/sub channel to broadcast the message to all
12: // other processes.
13: // As a result, every regular process (i.e., non-sequencer) uses a push
14: // and a sub socket.
15: var ps = zmq.socket('push');
16: var sb = zmq.socket('sub');
17: // The "local" object emulates the shared memory. Each one of its
18: // attributes is a shared variable.
19: var local = {};
20: // Ports being used by the sequencer process.
21: var seqPubPort = "9999";
22: var seqPullPort = "9998";
23: var url = "tcp://127.0.0.1:";
24: // Boolean variable that is set to true while a value is written and
25: // propagated to the remaining nodes.
26: var writing = false;
27:
28:
29: /*=========================================================================*/
30: /* PRIVATE FUNCTIONS                                                       */
31: /*=========================================================================*/
32:
33: /*************************************************************************/
34: /* join()                                                                */
35: /* The local process uses this function to connect its "subscriber"      */
36: /* socket to the "publisher" socket of the sequencer process and its     */
37: /* "push" to the "pull" socket to the sequencer.                         */
38: /*************************************************************************/
39: function join() {
```

```
40:     // Connect the push socket to the pull socket of the sequencer.
41:     ps.connect(url+seqPullPort);
42:     // Subscribe to everything.
43:     sb.subscribe("");
44:     // Connect the sub socket to the pub socket of the sequencer.
45:     sb.connect(url+seqPubPort);
46: }
47:
48: // The local process ID is initially set to -1.
49: var id = -1;
50:
51: /**************************************************************************/
52: /* myID(nid)                                                              */
53: /* Sets the local process ID and binds its publisher socket to its        */
54: /* intended port.                                                         */
55: /* Input args:                                                            */
56: /* - nid: Process identifier in the range 0..9                            */
57: /**************************************************************************/
58: function myID(nid) {
59:     // Checks whether the function has already been called.
60:     if (id!=-1)
61:         // If so, return without doing anything else.
62:         return;
63:     // Save the process identifier. "id" is global variable.
64:     id = parseInt(nid);
65:     // If the identifier has a wrong value, print an error message and
66:     // abort.
67:     if (isNaN(id) || id < 0 || id > 9) {
68:         console.log("ERROR: The node identifier should be a number " +
69:                     "in the range 0..9!!");
70:         process.exit();
71:     }
72: }
73:
74: /*========================================================================*/
75: /* PUBLIC FUNCTIONS                                                       */
76: /*========================================================================*/
77:
78: /**************************************************************************/
79: /* init(nodes)                                                            */
80: /* Initialises the "shared2" module, receiving the identifier of the      */
81: /* invoker local process.                                                 */
82: /* Input args:                                                            */
83: /* - id: Local process identifier.                                        */
84: /**************************************************************************/
85: exports.init = function(id) {
86:     join();
87:     myID(id);
88: }
89:
90: /**************************************************************************/
91: /* W(name,value)                                                          */
92: /* Writes the "value" value onto the "name" shared variable.              */
93: /* Input args:                                                            */
94: /* - name: String with the name of the shared variable being written.     */
95: /* - value: Value to be written.                                          */
96: /**************************************************************************/
97: exports.W = function (name, value) {
98:     // Propagate the <name,value> pair to the sequencer.
99:     ps.send([name, value, id]);
100: };
101:
102: /**************************************************************************/
103: /* R(name)                                                                */
104: /* Reads the current value of the "name" shared variable.                 */
105: /* Input args:                                                            */
106: /* - name: String with the name of the shared variable being read.        */
107: /* Return value:                                                          */
108: /* - Current value of such variable.                                      */
109: /**************************************************************************/
110: exports.R = function (name) {
111:     // Write a message on standard output describing this read action.
112:     process.stdout.write("R" + id + "(" + name + ")" + local[name] + ":");
113:     // Return the read value.
114:     return local[name];
115: };
116:
```

```
117: /*************************************************************************/
118: /* 'message' event handler                                               */
119: /* The messages being broadcast through the pub/sub channel consist in   */
120: /* an array with 3 slots: "name", "value" and "writer".                  */
121: /* We store the received value onto the local copy of the "name"         */
122: /* variable.                                                             */
123: /* In order to follow the trace, a "read" event is printed on screen.    */
124: /*************************************************************************/
125: sb.on('message', function (name, value, writer) {
126:     if (writer==id) {
127:         // Print a message on screen about its previous write action.
128:         process.stdout.write("W" + id + "(" + name + ")" + value + ":");
129:     }
130:     // Set the received value onto the local copy of the variable.
131:     local[name] = value;
132:     // Print a message on screen, emulating a local read.
133:     if (writer!=id) exports.R(name);
134: });
```

En este nuevo módulo (“shared2.js”), en lugar de que cada escritor difunda sus acciones al resto de procesos en el sistema, se pasa a que cada proceso envíe la escritura a un proceso “secuenciador”. El secuenciador redifunde esos mensajes recibidos a todos los procesos. El objetivo de este “protocolo de consistencia” es garantizar un mismo orden en la recepción de los valores escritos por todos los procesos que compartan la “memoria”. Esa memoria puede contener múltiples variables.

Para llegar a ese acuerdo en el orden de las escrituras necesitamos transformar la propia operación de escritura del módulo “shared2.js”. Ahora no escribirá en pantalla la “W” correspondiente cuando el proceso llame a ese método del módulo, sino que tendrá que esperar a que el mensaje correspondiente, una vez reenviado por el secuenciador, llegue de nuevo al proceso escritor. Es un protocolo “pesado” pues no permite al escritor consultar los valores que pretendía escribir hasta que estos hayan sido secuenciados.

Hay algunos modelos de consistencia que trabajan de esta manera: necesitan enviar mensajes a otros procesos antes de que se pueda leer el valor de una variable tras haber intentado escribir en ella. Por el contrario, en la actividad 1 no fue necesario esto.

Cuando en un modelo de consistencia se pueda leer el valor de una variable sin realizar ninguna espera se dice que es un ***modelo rápido*** de consistencia. La consistencia FIFO y la consistencia causal son modelos rápidos. Por el contrario, cuando para dar por terminada una escritura se necesita utilizar la red, entonces tendremos un ***modelo lento*** de consistencia. Los modelos de consistencia procesador y secuencial son ejemplos de modelos lentos.

Para utilizar este módulo “shared2.js” se ha desarrollado un programa “sequencer.js” para implantar el secuenciador. Se lista a continuación:

```
  1: /*************************************************************************/
  2: /*                                                                       */
  3: /* sequencer.js                                                          */
  4: /*                                                                       */
  5: /* This program implements a sequencer process. Such sequencer is used   */
  6: /* for implementing a total order propagation of the writes being        */
  7: /* generated by the other regular processes.                             */
  8: /*                                                                       */
  9: /*************************************************************************/
 10:
 11: var zmq = require("zmq");
 12:
 13: // Ports being used by the sequencer process.
```

```
14: var seqPubPort = "9999";
15: var seqPullPort = "9998";
16: var url = "tcp://127.0.0.1:";
17:
18: // Create and bind the sockets to their appropriate endpoints.
19: var pb = zmq.socket("pub");
20: pb.bindSync(url+seqPubPort);
21: var pl = zmq.socket("pull");
22: pl.bindSync(url+seqPullPort);
23:
24: // Handle the messages received in the pull port.
25: pl.on("message", function(name,value,writer) {
26:     // We only need to send the incoming message through the
27:     // publish socket.
28:     pb.send([name,value,writer]);
29: });
```

Para probar la funcionalidad del módulo “shared2.js” hay que lanzar en un primer paso al proceso secuenciador:

```
node sequencer &
```

...y después lanzar a todos los procesos que utilizarán la memoria “compartida”:

```
node proc2 1 & node proc2 2 & node proc2 3 &
```

Una salida obtenida en una ejecución de este tipo ha sido la siguiente:

```
1: W2(y)2:R2(y)3:R3(y)2:W3(y)3:R3(y)1:R2(x)11:R3(x)11:W1(x)11:R2(x)13:R3(x)13:W1(x)1
2: 3:W2(x)22:R3(x)22:R1(x)22:R2(x)17:R3(x)17:W3(x)33:R2(x)33:W1(x)17:R1(x)33:R2(x)39
3: :W3(x)39:R1(x)39:R3(x)19:W1(x)19:R3(x)26:R2(x)19:W2(x)26:R2(x)51:R1(x)51:W1(x)23:
4: R1(x)34:W3(x)51:R3(x)23:R3(x)34:R2(x)23:W2(x)34:W1(x)29:R3(x)29:R2(x)57:W2(x)38:R
5: 1(x)57:R1(x)38:W3(x)57:R3(x)38:R3(x)31:W3(x)69:R2(x)31:R2(x)69:W1(x)31:R1(x)69:R3
6: (x)46:R1(x)46:R2(x)87:W3(x)87:R3(x)37:R1(x)87:W1(x)37:R2(x)37:R3(x)58:W2(x)58:R1(
7: x)58:R2(x)41:R3(x)41:W1(x)41:W2(x)62:R1(x)62:R1(x)93:R2(x)93:R3(x)62:W3(x)93:R2(x
8: )43:W1(x)43:R3(x)43:W2(x)74:R3(x)74:R1(x)74:W3(x)111:R2(x)111:R1(x)111:R3(x)47:R2
9: (x)47:W1(x)47:W2(x)82:R3(x)82:R1(x)82:W3(x)123:R2(x)123:R1(x)123:R2(x)53:W1(x)53:
10: R3(x)53:W2(x)86:R3(x)86:W3(x)129:R2(x)129:R1(x)86:R1(x)129:R2(x)59:R3(x)59:W1(x)5
11: 9:R1(x)141:R2(x)141:W2(x)94:W3(x)141:R3(x)94:W1(x)11:R2(x)11:R3(x)11:R1(x)159:R2(
12: x)159:W3(x)159:R1(x)106:W2(x)106:R3(x)106:W1(x)13:R2(x)13:R3(x)13:...
```

Resuelva las siguientes cuestiones relacionadas con este ejemplo:

1. Justifique por qué pueden llegar a aparecer las lecturas de un determinado valor antes de mostrarse la escritura de ese mismo valor. Un ejemplo está resaltado en fondo amarillo dentro de la traza anterior, pero esta situación se da varias veces.
2. Ejecute las órdenes que acabamos de mencionar y, en base a la salida proporcionada (o en la que se ha mostrado en el fragmento de traza anterior) y el código de “shared2.js”, justifique qué modelos de consistencia se respetan al utilizar “shared2.js”.
3. Explique si “shared2.js” exige más o menos sincronización entre los procesos participantes que “shared1.js”. ¿Qué ejemplo implanta un modelo rápido de consistencia? ¿Cuál implanta un modelo lento de consistencia?
4. Justifique qué cambios deberían aplicarse sobre los programas mostrados en esta actividad para que el sistema resultante implantara consistencia caché pero no implantara consistencia secuencial. No es necesario que escriba un nuevo módulo ni que especifique qué líneas deberían modificarse en cada programa. Basta con describir las líneas generales.

ACTIVIDAD 3

En esta última actividad se ha vuelto a modificar el fichero "shared1.js" para generar otro módulo que proporcione soporte para el modelo de consistencia final (o consistencia eventual). Los cambios realizados pueden observarse en el fichero "shared3.js" y se han centrado en sustituir la sentencia "local[name] = value" que aparecía en las operaciones de escritura por otra sentencia "local[name] += parseInt(value)". Con ello, en vez de propagar el nuevo valor de la variable se propaga el valor con el que se incrementará, pues el operador utilizado ha pasado a ser "+=" en lugar de "=". Además, las operaciones de escritura se están difundiendo a todos los procesos. No tiene sentido aplicar la operación suma dos veces en el proceso escritor. Por ello, esa operación relacionada con la escritura solo se realiza en el momento de escribir y no se vuelve a aplicar cuando el escritor recibe el mensaje en el que difundía esa modificación a todos los demás procesos. Así no importará el orden de recepción de los mensajes. Si se reciben todos, los valores de las réplicas de cada variable serán consistentes.

A su vez, el fichero "proc1.js" ha sido sustituido por otro fichero "proc3.js" en el que se utiliza "shared3.js" en lugar de utilizar "shared1.js". Además, se ha modificado la gestión necesaria para terminar cada proceso "proc3", ampliando para ello su función terminate(). El código resultante es:

```
 1: /*********************************************************************/
 2: /* terminate()                                                       */
 3: /* Terminates the process, writing to screen the final value of the  */
 4: /* shared variable.                                                  */
 5: /*********************************************************************/
 6: function terminate() {
 7:     function a() {
 8:         console.log("P%d final value: %d", myID, sh.R("x"));
 9:         process.exit();
10:     }
11:     clearInterval(inter);
12:     setTimeout(a,100);
13: }
14:
15: function myActivity() {
16:     inter=setInterval(writeValue,10);
17: }
18:
19: ...
20: // Let the process start in two seconds.
21: setTimeout(myActivity, 2000);
22: // Terminate the process after some time.
23: setTimeout(terminate,3000);
```

Estos ficheros están disponibles en PoliformaT, en el ZIP mencionado en la primera página. Descárguelos y compruebe su funcionalidad lanzando al menos tres procesos "proc3" con diferentes identificadores.

Un ejemplo sería:

```
node proc3 1 & node proc3 2 & node proc3 3 &
```

Una salida obtenida en una ejecución de este tipo ha sido la siguiente:

```
1: W3(y)+3:R3(y)3:W1(y)+1:R1(y)1:W2(y)+2:R2(y)2:W3(x)+33:R3(x)33:R1(x)33:W1(x)+11:R1
2: (x)44:R2(x)33:W3(x)+39:R3(x)72:R3(x)83:R1(x)83:R2(x)44:R2(x)83:W3(x)+51:R3(x)134:
3: W1(x)+13:R1(x)96:R1(x)147:W2(x)+22:R2(x)105:R2(x)156:R3(x)156:R3(x)169:R1(x)169:R
4: 2(x)169:W3(x)+57:R3(x)226:W1(x)+17:R1(x)186:R1(x)243:W2(x)+26:R2(x)195:R2(x)212:R
5: 2(x)269:R1(x)269:R3(x)243:R3(x)269:W3(x)+69:R3(x)338:W2(x)+34:R2(x)303:R2(x)372:W
6: 1(x)+19:R1(x)288:R1(x)357:R1(x)391:R2(x)391:R3(x)372:R3(x)391:W2(x)+38:R2(x)429:R
7: 2(x)516:R2(x)539:W3(x)+93:R3(x)632:R1(x)632:R2(x)632:W2(x)+46:R2(x)678:W1(x)+29:R
8: 1(x)661:R3(x)678:R1(x)707:R3(x)707:R2(x)707:W3(x)+111:R3(x)818:R1(x)818:R2(x)818:
```

```
 9: W2(x)+58:R2(x)876:W1(x)+31:R1(x)849:R3(x)876:R2(x)907:R3(x)907:R1(x)907:W3(x)+123
10: :R3(x)1030:R1(x)1030:R2(x)1030:W1(x)+37:R1(x)1067:W2(x)+62:R2(x)1092:R1(x)1129:R3
11: (x)1067:R3(x)1129:R2(x)1129:W3(x)+129:R3(x)1258:R1(x)1258:W2(x)+74:R2(x)1332:R3(x
12: )1332:...
```

Como se observa en el fragmento resaltado, los procesos no siguen la misma secuencia de modificaciones, pues no tienen por qué aplicar todas las escrituras en un mismo orden.

Responda las siguientes cuestiones:

1. ¿Se obtiene el mismo valor final para la variable “x” en todos los procesos? ¿Por qué?
2. Para reforzar lo visto en la cuestión 1 de esta Actividad 3, modifique la función terminate() del programa proc3.js para que su código sea el siguiente:

```
 1: /*************************************************************************/
 2: /* terminate()                                                           */
 3: /* Terminates the process, writing to screen the final value of the      */
 4: /* shared variable.                                                      */
 5: /*************************************************************************/
 6: function terminate() {
 7:     clearInterval(inter);
 8:     console.log("P%d final value: %d", myID, sh.R("x"));
 9:     process.exit();
10: }
```

...y ejecute de nuevo tres procesos proc3 mediante...

```
node proc3 1 & node proc3 2 & node proc3 3 &
```

¿Qué ocurre ahora con los valores finales en cada proceso? ¿Por qué sucede eso?

3. Utilizando el código original de esta actividad... ¿Qué cree que ocurriría si un proceso proc3 (de entre los que hemos lanzado) fallase y lanzáramos uno nuevo que lo sustituyese mientras los otros dos permanecían en ejecución? ¿Terminarán todos con los mismos valores para la variable “x”? ¿Por qué?
4. Trate de describir el protocolo necesario para que la incorporación de un nuevo proceso (que sería una réplica del componente que está siendo emulado mediante el programa “proc3”) garantizase la consistencia final para esa nueva réplica. Esto es, para que el valor mantenido por esa nueva réplica de la variable “x” llegase a coincidir finalmente con el que mantienen todas las demás.