TSR

Aquest examen inclou 20 qüestions d'opció múltiple. Cadascuna d'elles només té una resposta correcta. Has de contestar en una altra fulla. Les respostes correctes aporten 0.5 punts a la teua qualificació. Les errònies descompten 0.167 punts.

TEORIA

1. Aquest NO ÉS un dels "aspectes rellevants" dels sistemes distribuïts:

a	Millorar l'eficiència de les aplicacions, dividint el problema a resoldre en tasques i executant cada tasca en un agent / ordinador diferent. Si una aplicació es dissenya com a distribuïda, les dades a processar han de repartir-se entre els seus processos. D'aquesta manera, la seua eficiència normalment millorarà. Per tant, aquest és un dels aspectes rellevants dels sistemes distribuïts.
b	Proporcionar transparència de fallades. Si una aplicació és executada per un sol agent, quan l'agent falle l'aplicació sencera fallarà. Per contra, en una aplicació distribuïda s'utilitzen múltiples agents que col·laboren entre si. Així, quan algun d'ells falle, els altres podran encara completar les seues respectives execucions i proporcionar un resultat vàlid. Per a fer això, els agents es repliquen. Per tant, les aplicacions distribuïdes poden proporcionar transparència de fallades i aquest és un dels seus principals avantatges.
c	Permetre la compartició de recursos, especialment d'aquells dispositius que resulten cars i poden accedir-se de manera remota. Aquest tipus de compartició no pot ocórrer en sistemes formats per un sol ordinador. Per això, aquesta és una de les raons principals que condueixen a l'ús de sistemes distribuïts.
d	Facilitar el desplegament de les aplicacions. Les aplicacions distribuïdes consten de diversos components, funcionant en múltiples ordinadors. Això complica la seua configuració i la resolució de dependències entre els components. A més, una vegada ja estiguen en execució, els seus usuaris esperen continuïtat en el servei. Aquesta continuïtat resulta difícil en cas que s'actualitzen els programes que composen l'aplicació. Aquestes tasques (instal·lació, configuració, resolució de dependències, actualitzacions) formen part del desplegament i resulten molt més complexes en les aplicacions distribuïdes que en les formades per un sol programa i una sola instància.

2. En els sistemes de computació en el núvol, la tecnologia de virtualització d'equips és un mecanisme bàsic per aquest model de servei:

a	SLA. Aquestes inicials corresponen als "acords de nivell de servei". Els SLA no són un exemple de model de servei. Són un acord entre el proveïdor de serveis i el client sobre el nivell de qualitat que haja de garantir-se. Els models de servei indiquen quin és el servei principal que proporciona un proveïdor de sistemes de computació en el núvol.

b	SaaS. SaaS sí és un model de servei per als sistemes de computació en el núvol. No obstant això, el seu principal objectiu és facilitar serveis "de programari" als seus clients. Per tant, se centra en el disseny i l'arquitectura d'aquestes aplicacions distribuïdes i la seua principal preocupació no és la virtualització de la infraestructura.
ℂ	IaaS. La virtualització de la infraestructura és el mecanisme bàsic pel qual han de preocupar-se els proveïdors d'infraestructura. Com el model de servei IaaS està enfocat a la provisió d'infraestructura, la virtualització és la tecnologia clau en aquest model.
d	*Middleware* de comunicacions (basat en missatges). Els nivells "middleware" no són exemples de models de serveis per a la computació en el núvol.

3. En el model SaaS (per a sistemes de computació en el núvol) aquesta afirmació és certa:

a	Els seus serveis poden accedir-se localment, sense necessitat d'utilitzar la xarxa, emprant virtualització. El model SaaS no gestiona accessos a serveis "locals". Els models de servei de computació en el núvol s'han definit per a la provisió de serveis a usuaris remots en tots els casos. Per tant, la primera part de l'afirmació no té sentit. A més, la virtualització de la infraestructura no és el principal objectiu d'aquest model.
b	Les seues interaccions client-servidor han de basar-se en *un middleware* de comunicacions asincròniques basat en missatges. En aquest model no s'obliga al fet que les comunicacions estiguen orientades a missatges ni al fet que siguen asincròniques.
ℂ	Proporciona serveis de *programari* distribuïts als seus clients, generalment amb un model de pagament per ús. Aquesta afirmació pot considerar-se una definició breu del model.
d	Els usuaris dels serveis decideixen de quina manera es desplegaran els programes. El desplegament dels serveis mai hauria de ser una responsabilitat dels usuaris. En el model SaaS es respecta aquesta restricció. Per això, els usuaris d'un servei no participen en aquestes tasques del cicle de vida dels serveis.

4. El Tema 2 recomana el paradigma de programació asincrònica perquè aquest paradigma...

a	...aconsegueix que el desplegament d'aplicacions siga trivial. Els paradigmes de programació no tenen un efecte important sobre les tasques de desplegament. El desplegament inclou aquelles etapes del cicle de vida dels programes que seguisquen al seu desenvolupament (instal·lació, configuració, activació, actualització, escalat, desactivació, eliminació...) i moltes d'aquestes etapes no depenen del model de programació utilitzat per a desenvolupar l'aplicació.

b	…està basat en esdeveniments i assegura l'execució atòmica de cada acció. Ambdues característiques es respecten en aquest paradigma de programació. A més, aquestes característiques són interessants, ja que l'execució atòmica evita les condicions de carrera i l'orientació a esdeveniments proporciona una correspondència directa entre el que s'especifique en un algorisme i el comportament real de les accions de l'aplicació.
c	…proporciona transparència de fallades. La transparència de fallades solament pot proporcionar-se quan els processos de l'aplicació estiguen replicats. L'ús de replicació ha de preveure's durant l'etapa de disseny de l'aplicació i el nombre de rèpliques variarà dinàmicament en funció d'alguns factors (per exemple, el nivell actual de càrrega). El paradigma de programació no influeix en cap d'aquests aspectes.
d	…utilitza *proxies* inversos i, a causa d'això, és altament escalable. Els *proxies* inversos són un exemple de mecanisme de propagació de missatges i ús d'una memòria "cache" (quant a la seua proximitat i rapidesa d'accés) en la gestió de resultats. Aquests aspectes no estan considerats en la definició de la programació asincrònica.

5. Si considerem els aspectes de la sincronia vistos en el Tema 2, és cert que...:

a	Els rellotges lògics generen processos sincrònics. Per a afirmar que els processos utilitzats en una aplicació o algorisme són sincrònics, aquests han d'avançar les seues accions de manera sincronitzada: en cada pas o etapa, tot procés de l'algorisme ha de completar una acció. Els rellotges lògics no aconsegueixen aquest tipus de sincronització forta. Amb ells es pot etiquetar cada acció realitzada per a determinar posteriorment si hi ha hagut dependències d'ordre entre algunes de les accions realitzades pels processos. En cap moment condicionen quan es podrà iniciar una nova acció o quant ha de tardar-se a finalitzar-la.

b	L'ordre (sincrònic) dels missatges limita el temps de propagació dels missatges. Quan puga limitar-se el temps de transmissió o propagació dels missatges es podrà parlar de "canals sincrònics". El "ordre sincrònic dels missatges" es refereix a una altra cosa. Concretament, al fet que els missatges transmesos en l'aplicació o algorisme respectaran cert ordre d'entrega; per exemple, FIFO, causal, total... Per a fer això, els nodes utilitzaran "buffers" de recepció i algun protocol de comunicacions que permeta respectar aquest ordre. Aquests protocols no necessiten utilitzar canals sincrònics. Per exemple, per a garantir ordre FIFO n'hi ha prou que cada procés emissor numere els missatges que ell envia. Els receptors hauran de respectar l'ordre seqüencial de numeració a l'hora de lliurar-los. Si s'utilitzaren canals sincrònics i no es numeraren els missatges, l'ordre FIFO podria no respectar-se.
c	Els processos sincrònics avancen en passos. En cada pas, tot procés completa una acció. Aquesta és una definició concisa i vàlida del terme "procés sincrònic".
d	La comunicació sincrònica requereix que els canals mantinguen els missatges enviats fins que els receptors puguen acceptar-los. Aquesta és la definició de "comunicació persistent". La comunicació sincrònica es dóna quan l'emissor es bloqueja fins que reba algun reconeixement sobre el correcte lliurament del missatge que acaba d'enviar. Aquests dos tipus de comunicació consideren aspectes diferents. Per tant, l'afirmació és falsa.

6. Aquestes afirmacions relacionen el *middleware* amb els estàndards. Quina és falsa?

a	L'ús d'estàndards permet que els *middleware* i les implementacions d'agents realitzades per diferents empreses siguen interoperables. Aquesta afirmació és certa. Un dels objectius dels estàndards és facilitar la interoperabilitat.
b	L'ús d'estàndards proporciona una interfície d'alt nivell en *els middleware*. Així, les tasques de programació resulten més senzilles. Cert. Les interfícies proporcionades per *un middleware* solen ser d'alt nivell i estar basades en les solucions definides en alguns estàndards per a certs problemes.
c	Les APIs proporcionades per sistemes *middleware* no sempre són estàndard. ZeroMQ n'és un exemple. Cert. ZeroMQ és un exemple de *middleware* per a comunicació asincrònica. Hi ha un estàndard en aquest camp: AMQP, però ZeroMQ no el respecta.
d	Els sistemes *middleware* no han de respectar cap estàndard, ja que els estàndards només es defineixen per als elements interns dels sistemes operatius. Fals. Els estàndards no estan limitats als elements interns dels sistemes operatius. Els sistemes *middleware* se situen sobre el sistema operatiu i solen respectar alguns estàndards, principalment quan la interoperabilitat siga un dels seus objectius.

7. Quin dels següents elements de comunicació pot considerar-se un exemple de *middleware*?

a	El protocol IP. Tot *middleware* ha de situar-se sobre el sistema operatiu. El protocol IP està en el nivell de xarxa (nivell 3) en una arquitectura de comunicacions basada en nivells. Els

	nivells entre 2 i 4, inclusivament, són gestionats pel sistema operatiu. Per tant, aquest protocol està per baix del *middleware* i no pot ser un exemple d'ell.
b	Un servei de noms distribuït. Aquest tipus de servei està per damunt del sistema operatiu i facilita una base per a proporcionar transparència d'ubicació en un sistema distribuït. Els seus serveis són utilitzats per aplicacions que estan conceptualment sobre ell en una arquitectura estructurada en nivells. Per això, pot considerar-se un exemple de *middleware*.
c	TCP. TCP és un protocol de transport. Per tant, pertany al nivell 4 i està gestionat pel sistema operatiu. Igual que en l'apartat "a", és un protocol situat en nivells inferiors als ocupats pel *middleware*.
d	El servidor APACHE. Aquest és un exemple de "servidor web". Des del punt de vista de les arquitectures de comunicacions, pertany al nivell d'aplicació. Per això, està situat sobre el *middleware*.

8. En l'àmbit dels sistemes *middleware*, quins són els problemes dels sistemes d'objectes distribuïts quan són comparats amb els sistemes de missatgeria?

a	El seu acoblament (potencialment alt) pot conduir a bloquejos perllongats quan algun recurs compartit és utilitzat concurrentment per molts agents. Cert. Els sistemes d'objectes faciliten transparència d'ubicació. Tota invocació ofereix la imatge de ser local, però pot accedir objectes remots. Aquests objectes remots necessitaran mecanismes de control de concurrència per a gestionar múltiples invocacions simultànies, des de diferents clients. Aquests mecanismes podran bloquejar l'activitat dels agents durant intervals perllongats.
b	No proporcionen transparència d'ubicació. Fals. Les interaccions entre *proxies* i esquelets faciliten transparència d'ubicació perquè el *proxy* ofereix la mateixa interfície que l'objecte remot i oculta l'intercanvi de missatges.
c	Faciliten un baix nivell d'abstracció, complicant els programes resultants. Al contrari: ofereixen un nivell d'abstracció alt. Les aplicacions es poden desenvolupar fàcilment en aquest model.
d	El seu comportament és excessivament asincrònic, i per això no poden depurar-se fàcilment. Les aplicacions orientades a objectes usen interacciones client-servidor, per la qual cosa el client (emissor inicial) roman bloquejat fins a rebre la resposta. Aquestes interaccions, com acabem d'explicar, serien un exemple de comunicació sincrònica. Per tant, el seu comportament no pot qualificar-se com a asincrònic.

SEMINARIS

9. Considere's aquest programa:

```
var fs=require('fs');
if (process.argv.length<5) {
  console.error('More file names are needed!!');
  process.exit();
}
var files = process.argv.slice(2);
var i=-1;
do {
  i++;
  fs.readFile(files[i], 'utf-8', function(err,data) {
      if (err) console.log(err);
      else console.log('File '+files[i]+': '+data.length+' bytes.');
  })
} while (i<files.length-1);
console.log('We have processed '+files.length+' files.');
```

Aquesta afirmació és certa si assumim que cap error avorta la seua execució i es passen suficients noms de fitxer com a arguments des de la línia d'ordres:

a	A causa de l'asincronia del *callback* emprat en readFile(), aquest programa no mostra en cada iteració el nom i longitud correctes per a cada fitxer. El programa mostra la longitud correcta en cada iteració, però no el nom. La longitud es calcula utilitzant un dels arguments del *callback* ("data"). En aquest argument es rep el contingut del fitxer corresponent, per la qual cosa el càlcul sempre serà vàlid. No obstant això, el nom es pren directament del vector "files", utilitzant per a fer això el valor actual de la variable "i". Com readFile() és una operació asincrònica, els *callbacks* aniran sent invocats una vegada el bucle do..while haja finalitzat totes les seues iteracions. En aquell moment, el valor de "i" serà igual a "file.length". Desafortunadament l'element "files[file.length]" té un valor "undefined" perquè en aquesta component no s'ha arribat a guardar cap nom. Per exemple, si es van passar tres noms de fitxer "a", "b" i "c" com a arguments, cadascun dels noms estarà en les components "files[0]", "files[1]" i "files[2]", respectivament, però "files.length" serà 3 i "files[3]" no guardarà cap valor.
b	Mostra el nom i grandària de cada fitxer rebut com a argument. Serà incapaç de mostrar el nom, com ja s'ha explicat en l'apartat "a".
c	Mostra "We have processed 0 files" com el seu primer missatge en pantalla. Fals. Sí que arriba a mostrar-se un missatge "We have processed..." en començar l'execució, però no visualitza "0 files" en la seua part final. En el seu lloc, es mostra el nombre correcte de noms de fitxer rebuts com a arguments. Observe's que "files.length" ja està definit aleshores i el seu valor ha de ser superior a 2 perquè la segona instrucció del programa exigeix que process.argv.length siga superior a 4 i posteriorment, en la sisena línia, s'eliminen les dues primeres components del vector ("node" i el nom del programa) per a assignar el vector resultant a "files".
d	Descarta alguns dels noms de fitxer proporcionats com a arguments després dels elements "node nom-programa". No. Com ja hem explicat en l'apartat anterior, s'utilitza el mètode "slice()" per a eliminar les dues primeres components de "process.argv[]" quan el seu contingut és copiat en la variable "files". Aquestes dues components no contenen cap dels noms de fitxer utilitzats com a arguments.

10. La següent afirmació sobre el programa de la qüestió anterior és certa:

a	Necessita diversos torns per a completar la seua execució, ja que cada fitxer a llegir necessita un torn per al *seu callback*. Les lectures són asincròniques. Per això, quan cadascuna finalitza, crida al seu *callback* associat. L'execució del *callback* es fa dins del context d'un fil independent, i cadascun d'aquests fils utilitzarà un torn d'execució diferent en el *runtime* de "node".
b	L'increment de la "i" (instrucció "i++") està situat incorrectament. Hauria d'estar dins del *callback*. No. Si es fera d'aquesta manera, el bucle mai finalitzaria i s'iniciarien infinites lectures del primer fitxer el nom del qual s'ha passat com a argument en la línia d'ordres.
c	Aquest programa mostra un error i finalitza si s'han passat menys de cinc noms de fitxer com a arguments. No. Això només ocorre quan es passen menys de tres noms de fitxer. Si es passen tres o quatre noms, l'execució continua amb normalitat i no mostra cap error associat al nombre d'arguments facilitats.
d	Mostra la mateixa grandària en totes les iteracions. Es necessita una clausura per a evitar aquest comportament incorrecte. No. Com ja es va explicar en la qüestió nou, la longitud de cada fitxer es mostra correctament en cada iteració.

11. Respecte als algorismes d'exclusió mútua del Seminari 2, aquesta afirmació és certa:

a	L'algorisme de servidor central gestiona correctament aquelles situacions en les quals el servidor central falla. Fals. Si falla el servidor central, l'algorisme no podrà continuar. Els algorismes centralitzats solen tenir aquest problema.

b	L'algorisme d'anell virtual unidireccional no perd *el token* si el procés actualment en la secció crítica falla. Fals. El procés que actualment execute la secció crítica podrà fer això perquè manté el *token*. Si fallara, el *token* deixaria d'existir.
c	L'algorisme de difusió amb rellotges lògics usa menys missatges que l'algorisme de difusió basat en quòrums. No. Els quòrums van ser introduïts en els algorismes de difusió per a reduir la quantitat de missatges a utilitzar. En lloc d'enviar un missatge a cada participant, en ells n'hi ha prou amb enviar un missatge a cada membre del quòrum. Cada quòrum no inclou a tots els processos que utilitzen l'algorisme.
d	L'algorisme de difusió amb rellotges lògics compleix les tres condicions de correcció del problema d'exclusió mútua. Cert. Satisfà seguretat, vivacitat i ordre causal. Aquestes són les tres condicions de correcció per a aquest problema.

12. Considerant aquest programa i sabent que no genera cap error...

```
var ev = require('events');
var emitter = new ev.EventEmitter;
var num1 = 0;
var num2 = 0;
function myEmit(arg) { emitter.emit(arg,arg) }
function listener(arg) {
   var num=(arg=="i1"?++num1:++num2);
   console.log("Event "+arg+" has happened " + num + " times.");
   if (arg=="i1") setTimeout( function() {myEmit("i2")}, 3000 );
}

emitter.on("i1", listener);
emitter.on("i2", listener);
setTimeout( function() {myEmit("i1")}, 2000 );
```

La següent afirmació és certa:

a	L'esdeveniment “i1” ocorre una sola vegada, dos segons després d'iniciar-se el procés. Sí. L'última línia programa la generació d'aquest esdeveniment dos segons després. El *listener* para “i1” incrementa “num1” i mostra “Event i1 has happened 1 times.” en la pantalla. Llavors, es programa la generació d'“i2” tres segons després. Però en *el listener* ja no es fa cap altre “setTimeout()”si l'esdeveniment gestionat és “i2”. Per això, “i1” ja no tornarà a generar-se.
b	L'esdeveniment “i2” no ocorre mai. Fals. Com hem explicat en l'apartat “a”, l'esdeveniment “i2” ocorre una vegada.
c	L'esdeveniment “i2” es dóna periòdicament, cada tres segons. Fals. Com hem explicat en l'apartat “a”, l'esdeveniment “i2” ocorre només una vegada.
d	L'esdeveniment “i1” es dóna periòdicament, cada dos segons. Fals. Com hem explicat en l'apartat “a”, l'esdeveniment “i1” ocorre només una vegada.

13. Considerant el programa de la qüestió anterior, la següent afirmació és certa:

a	El primer esdeveniment “i2” ocorre tres segons després d'iniciar-se el procés. No. Succeeix cinc segons després d'iniciar-se el procés.

b	Com tots dos esdeveniments utilitzen el mateix *listener*, tots dos mostren missatges amb exactament el mateix contingut quan ocorren. No. Tots dos esdeveniments empren el mateix *listener*, però reben un argument. L'argument permet que el *listener* es comporte de manera diferent per a cada esdeveniment.
c	El primer esdeveniment "i2" ocorre dos segons després del primer esdeveniment "i1". No. Aquest termini dura tres segons.
d	Cap dels esdeveniments ocorre dos o més vegades. Cert. Tots dos ocorren una sola vegada.

14. En ØMQ, el patró de comunicacions REQ-REP es considera sincrònic perquè:

a	Tots dos *sockets* estan connectats o han fet un "bind()" sobre el mateix URL. Fals. Per exemple, un patró de comunicació PUSH-PULL també exigeix que tots dos *sockets* utilitzen un mateix URL, però el canal resultant és asincrònic. Per tant, aquest aspecte no és una característica dels canals sincrònics.
b	Tots dos *sockets* són bidireccionals. Fals. Els *sockets* ROUTER i DEALER són també bidireccionals i permeten definir un patró de comunicacions. No obstant això, ROUTER i DEALER són asincrònics. Per tant, la bidireccionalitat no està lligada a la sincronia.
c	El *socket* REP utilitza una operació sincrònica per a manejar els missatges rebuts. Fals. Tots els *sockets* ZeroMQ usen un *listener* (és a dir, un tipus de *callback*) per a gestionar la recepció de missatges. Els *listeners* tenen un comportament asincrònic.
d	Després d'enviar un missatge M, tots dos *sockets* no poden transmetre un altre missatge fins que s'haja rebut una resposta a M (REQ) o una nova petició (REP). Cert. Aquest comportament obliga al fet que tant REP com REQ s'utilitzen de manera sincrònica, seguint un ordre seqüencial i bloquejant en la transmissió dels missatges. Així, un servidor no pot transmetre una nova resposta fins que reba una nova petició. De la mateixa manera, un client no pot transmetre una nova petició mentre no reba una resposta per a la petició actual.

15. Considerant aquests dos programes NodeJS...

```
// server.js
var net = require('net');
var server = net.createServer(
 function(c) {//'connection' listener
  console.log('server connected');
  c.on('end', function() {
   console.log('server disconnected');
  });
  c.on('data', function(data) {
   console.log('Request: ' +data);
   c.write(data+ 'World!');
  });
});
server.listen(9000);
```

```
// client.js
var net = require('net'); var i=0;
var client = net.connect({port:
  9000}, function() {
    client.write('Hello ');
  });
client.on('data', function(data) {
  console.log('Reply: '+data);
  i++; if (i==1) client.end();
});
client.on('end', function() {
  console.log('client ' +
    'disconnected');
});
```

Aquesta afirmació és certa:

a	El servidor acaba després d'enviar la seua primera resposta al primer client. Fals. No hi ha cap instrucció que provoque la finalització del servidor en el seu programa. Per tant, el servidor no finalitza de manera voluntària.

b	El client no acaba mai. Fals. Quan el client reba la seua primera resposta, incrementarà el valor de "i". Llavors, comprovarà si "i" val 1 (i així és). En aquest cas, tanca la seua connexió. Quan la connexió es tanca el client gestiona l'esdeveniment "end" i a partir d'aquest moment la connexió desapareix. Amb això ja no pot haver-hi cap esdeveniment pendent per al qual el procés client tinga un *listener* associat i per això el procés finalitza en aquest moment. Això no causa la finalització del servidor, ja que el servidor encara pot gestionar altres connexions amb altres clients (tant en aquest mateix moment com més tard).
c	El servidor pot gestionar múltiples connexions. Cert. L'operació createServer() utilitza un gestor de connexió com el seu *callback*. D'aquesta manera, pot gestionar diverses connexions. Per a cada connexió hi haurà *listeners* per als seus esdeveniments 'data' (recepció de missatges) i 'end' (tancament de la connexió).
d	El client no pot connectar amb el servidor. Fals. Si el client i el servidor són llançats en un mateix ordinador, com tots dos utilitzen el mateix port local (9000) podran comunicar-se sense excessius problemes. El servidor atén noves connexions en aquest port, perquè ha utilitzat per a fer això l'operació 'listen', mentre que el client es connecta al port amb l'operació 'connect'.

16. Els algorismes d'elecció de líder (del Seminari 2)...

a	...necessiten l'execució prèvia d'un algorisme d'exclusió mútua, ja que la identitat del líder es guarda en un recurs compartit i només pot modificar-la un procés. Fals. Cada procés guarda la identitat del líder en una variable pròpia. Per ser pròpia, cap d'ells pot observar ni accedir directament al valor mantingut pels altres processos. Per tant, no es necessita cap algorisme d'exclusió mútua per a gestionar la identitat.
b	…necessiten consens entre tots els processos participants: tots han de triar un mateix líder. Cert. Aquest és l'objectiu d'aquests algorismes, que el valor triat siga acceptat per tots.
c	…no necessiten identitats úniques per a cada procés. Fals. Es necessita utilitzar identitats úniques perquè cada procés puga distingir-se dels altres.
d	...han de respectar ordre causal. Fals. La comunicació entre processos no necessita ordre causal en els algorismes d'elecció de líder presentats en el Seminari 2. Això va ser un requisit per a les solucions al problema d'exclusió mútua si es pretenia assegurar equitat entre els participants.

17. Es vol desenvolupar un programa d'elecció de líder en NodeJS i ØMQ, utilitzant el primer algorisme del Seminari 2: el d'anell virtual. Per a fer això, selecciona la millor opció d'entre les següents:

a	Cada procés usa un *socket* REQ per a enviar missatges al seu successor en l'anell i un *socket* REP per a rebre missatges del seu predecessor. Fals. Aquest algorisme utilitza comunicació unidireccional. Això significa que cada procés rep missatges del seu predecessor i els envia (quan es necessite) al seu

	successor. Els sockets REQ i REP no poden utilitzar-se en aquest tipus de comunicació. Són bidireccionals i no permeten l'enviament d'un segon missatge mentre no es reba un altre missatge entre tant. Per això, els sockets REQ es bloquejarien en tractar d'enviar un segon missatge i serien incapaços de transmetre res més.
b	Cada procés usa un *socket* ROUTER per a enviar missatges al seu successor en l'anell i un *socket* DEALER per a rebre missatges del seu predecessor. Fals. Encara que tots dos són asincrònics, el socket ROUTER no pot enviar un missatge a un procés A mentre no haja rebut algun missatge des d'A. Quan reba aquest primer missatge, el socket ROUTER passarà com a primer segment al procés receptor la identitat de la connexió establida amb A. Una vegada coneguda la identitat, podrà utilitzar-la per a enviar missatges a A per la connexió. Aquest algorisme utilitza comunicació estrictament unidireccional. Això evita que cada socket ROUTER arribe a conèixer la identitat del seu procés successor. Sense ella, no es podrà enviar cap missatge.
c	Cada procés usa un *socket* SUB per a enviar missatges al seu successor en l'anell i un *socket* PUB per a rebre missatges del seu predecessor. Fals. Encara que els sockets PUB i SUB realitzen comunicació unidireccional, el socket SUB no pot ser utilitzat per a enviar missatges i el socket PUB no pot utilitzar-se per a rebre'ls.
d	Cada procés usa un *socket* PUSH per a enviar missatges al seu successor en l'anell i un *socket* PULL per a rebre missatges del seu predecessor. Cert. Aquesta combinació és la més senzilla per a implantar comunicació unidireccional.

18. Es vol desenvolupar un programa d'elecció de líder en NodeJS i ØMQ, utilitzant el segon algorisme (intimidador o "bully") del Seminari 2. Per a suportar els missatges "elecció" (per a preguntar als millors candidats sobre la seua vivacitat) i "resposta" (a un "elecció" previ, confirmant la vivacitat), una alternativa viable podria ser, assumint N processos:

a	Un *socket* REQ connectat per a enviar "elecció" als altres N-1 processos i rebre les seues "respostes" i un *socket* REP lligat a un port local, per a rebre "eleccions" i enviar "resposta". Fals. Els missatges "elecció" i "resposta" es poden considerar missatges de petició i resposta, respectivament. A primera vista, aquest patró podria ser suportat per sockets REQ i REP. No obstant això, en aquest apartat s'indica que el REQ d'un procés hauria d'estar connectat a tots els REP de tots els altres processos. En aquest cas, l'emissor no tindria cap control a l'hora d'especificar a quin procés li està enviant el missatge "elecció" actual. L'algorisme indica que aquests missatges han d'enviar-se als possibles candidats (és a dir, tots aquells amb un identificador major que el de l'emissor) i aquesta selecció no pot dur-se a terme amb un únic socket REQ. Si connectem un socket REQ a múltiples sockets REP, els seus missatges de distribueixen seguint un ordre circular. Únicament el socket ROUTER permet triar quina de les connexions establides pel socket haurà d'utilitzar-se en cada enviament. Per a fer això es requereix a més que prèviament s'haja rebut algun missatge per aquesta mateixa connexió.

b	Un únic *socket* DEALER per a enviar "elecció" i "resposta" als altres N-1 processos. El mateix socket s'utilitzarà per a rebre els missatges dels altres. A l'hora d'enviar, els sockets DEALER ofereixen el mateix problema que els sockets REQ: tots dos utilitzen un torn circular entre les seues connexions. Per això, no hi ha forma de seleccionar a quins altres processos enviarem el missatge "elecció". Això fa que aquesta opció no siga viable.
c	N-1 *sockets* PUSH per a enviar "elecció" i "resposta" als altres N-1 processos. Un únic *socket* SUB per a rebre els missatges dels altres. Els sockets SUB no han de ser utilitzats per a rebre els missatges enviats mitjançant sockets PUSH. Hauria de ser un socket PULL en lloc de SUB.
d	Un únic *socket* PULL per a rebre missatges. N-1 *sockets* PUSH connectats als PULL dels altres processos, per a enviar "elecció" i "resposta" quan es necessite. Aquest esquema sí que arribaria a funcionar. Tenint N-1 sockets PUSH es podrà seleccionar a quin altre procés li enviem el missatge "elecció". Per a fer això el programa corresponent hauria de saber quin socket correspon a cada altre procés (una forma senzilla d'aconseguir-ho és mantenir un vector de sockets, indexat amb l'identificador de procés i forçar al fet que tots els missatges "elecció" incloguen l'identificador del seu emissor, perquè així el receptor sàpia quin socket utilitzar per a enviar la "resposta" associada). Mitjançant un sol socket PULL en cada participant es poden rebre els missatges enviats pels altres processos i dirigits a aquest receptor.

19. Quin és el tipus de *socket* ØMQ que utilitza múltiples cues d'enviament?

a	El tipus PUB, per a gestionar les seues difusions. No. Solament hi ha una cua d'enviament en aquest tipus de socket. Els missatges difosos prèviament no arriben a ser rebuts per aquells subscriptors que es connecten més tard.

b	El tipus PUSH, per a gestionar múltiples operacions send() asincròniques. Fals. Hi ha una única cua d'enviament en aquest tipus de sockets. Cada missatge només es propaga a un receptor. En cas de múltiples connexions, se segueix un torn circular per als enviaments.
c	El tipus REQ, en cas d'estar connectat a múltiples *sockets* REP. Fals. Hi ha una única cua d'enviament en aquest tipus de sockets. Cada missatge només es propaga a un receptor. En cas de múltiples connexions, se segueix un torn circular per als enviaments.
d	El tipus ROUTER, utilitzant una cua d'enviament per a cada connexió. Cert. Cada cua d'enviament s'associa a un identificador de connexió diferent. A causa d'això, en les operacions send(), els sockets ROUTER necessiten (com el primer segment del missatge a enviar) la identitat de la connexió o cua d'enviament per la qual ha de transmetre's el missatge.

20. Si considerem aquests programes...

```
//client.js
var zmq=require('zmq');
var rq=zmq.socket('req');
rq.connect('tcp://127.0.0.1:8888');
rq.connect('tcp://127.0.0.1:8889');
for (var i=1; i<=100; i++) {
  rq.send(''+i);
  console.log("Sending %d",i);
}
rq.on('message',function(req,rep){
  console.log("%s: %s",req,rep);
});
```

```
// server.js
var zmq = require('zmq');
var rp = zmq.socket('rep');
var port = process.argv[2] || 8888;
rp.bindSync('tcp://127.0.0.1:'+port);
rp.on('message', function(msg) {
  var j = parseInt(msg);
  rp.send([msg,(j*3).toString()]);
});
```

...i suposem que hem iniciat un client i dos servidors amb aquesta ordre:
$ node client & node server 8888 & node server 8889 &
La següent afirmació és certa:

a	Un servidor rep totes les sol·licituds amb valor parell per a "i" i l'altre rep totes les sol·licituds amb valor imparell per a "i". Cert. Quan un socket REQ (o DEALER o PUSH) es connecte a múltiples URL, utilitzarà una gestió circular per a distribuir els missatges enviats a través d'elles. En aquest cas, el socket REQ s'ha connectat a dos servidors diferents. Com els missatges es numeren consecutivament, un dels servidors rep els missatges amb nombres imparells i l'altre aquells amb nombres parells.

b	Cada servidor rep, gestiona i contesta les 100 sol·licituds. Així, el client rep i mostra 200 respostes. No. Els missatges es reparteixen entre tots dos. Això implicarà que un servidor reba 50 missatges i l'altre els altres 50.
c	Algunes sol·licituds inicials es perden, perquè el client ha sigut iniciat abans que començara el primer servidor. No. Els missatges no poden enviar-se mentre les connexions corresponents no s'hagen establit correctament. Per això, cap missatge arriba a perdre's amb aquest tipus de sockets.
d	Si un dels servidors falla durant l'execució, el client i l'altre servidor gestionaran sense interrompre's les altres sol·licituds i respostes. No. Després de la fallada d'un dels servidors, el client romandrà esperant la seua resposta de manera indefinida. Això implica que ja no podrà transmetre cap missatge més a través de la xarxa. Per tant, les altres sol·licituds no arribaran a transmetre's. Això provocarà que les seues respostes tampoc siguen generades. El client arribarà a acabar el seu bucle d'enviaments, però aquests missatges romandran indefinidament en la cua d'enviament. El seu bloqueig es percep al no mostrar cap altre missatge per pantalla sobre la recepció de respostes.