TSR: Primer Parcial

L'examen consta de 20 preguntes d'opció múltiple. En cada cas només una de les respostes és correcta. S'ha de respondre en una fulla a part. En cas de resposta correcta, aquesta aportarà 0.5 punts. En cas d'error, la contribució és negativa: -0.167.

TEORIA

1. Considerant el Tema 1, aquestes afirmacions descriuen correctament alguns aspectes dels sistemes distribuïts:

a	Tot sistema concurrent és un sistema distribuït. Fals. Tots els sistemes distribuïts són exemples de sistemes concurrents, però no tots els sistemes concurrents són distribuïts. Per exemple, un programa Java que use dos fils (un per a interactuar amb l'usuari i un altre per a accedir a fitxers) generarà un procés concurrent, però aquest procés no és una aplicació distribuïda.
b	El servei de correu electrònic és un exemple de sistema distribuït. Vertader. En un servei de correu electrònic hi ha diferents tipus d'agents (servidors, clients,...) i aquests agents es despleguen en múltiples ordinadors i necessiten col·laborar entre si per a gestionar la redacció, enviament, encaminament, emmagatzematge i entrega dels correus.
c	Els agents en un sistema distribuït no poden compartir cap recurs ja què estan situats en ordinadors diferents. Fals. Alguns recursos poden compartir-se entre els agents. En cas contrari no s'hauria necessitat cap algorisme distribuït d'exclusió mútua (dels quals hem vist múltiples exemples en el Seminari 2).
d	El programador d'una aplicació distribuïda no necessita preocupar-se sobre la tolerància a fallades, perquè ja està garantida implícitament pel sistema distribuït. Fals. Per a proporcionar tolerància a fallades es necesita replicar els components. La replicació no es facilita automàticament en tots els "sistema distribuïts". La gestió de la replicació sol ser responsabilitat de les aplicacions distribuïdes (alguns tipus particulars de "middleware" poden cooperar en aquesta gestió).

2. Una de les raons per a dir que la Viquipèdia és una aplicació distribuïda escalable és...

a	Des de la seua primera edició, s'ha implantat en el núvol seguint el model de servei SaaS. Fals. Les primeres edicions de la Viquipèdia van ser generades en 2001, abans que el model de servei SaaS començara a popularitzar-se. A més, en aqueixes edicions la Viquipèdia podia utilitzar uns pocs ordinadors per a proporcionar els seus serveis.
b	És un sistema LAMP, i tots els sistemes d'aquest tipus són escalables. Fals. Un sistema LAMP no té per què ser escalable. Hauria de complementar-se amb una estratègia adequada de gestió de memòries cau, interacció "*stateless*" amb els clients i replicació de components per a millorar la seua escalabilitat.
c	Utilitza interacció P2P i això millora la seua escalabilitat. Fals. Usa interacció client-servidor.
d	Utilitza *proxies* inversos (com a memòries cau) i replicació de components. Vertader. Ambdues tècniques milloren l'escalabilitat.

3. L'objectiu principal del model de servei PaaS és…

a	Automatitzar la configuració, desplegament i actualització de serveis distribuïts i la

	seua adaptació a càrregues variables. Vertader. Aquests són els seus objectius principals.
b	Automatitzar la provisió d'infraestructura. Fals. Aquest és l'objectiu principal del model de servei IaaS.
c	Proporcionar serveis distribuïts amb un model de pagament per ús ("pay-as-you-go"). Fals. Aquest és l'objectiu principal del model de servei SaaS.
d	Gestionar dades persistents amb un model de pagament per ús ("pay-as-you-go"). Fals. Aquest és l'objectiu principal del model de servei DBaaS (subconjunt de l'IaaS).

4. El tema 2 proposa un model de sistema distribuït senzill perquè aquest model…

a	…garanteix persistència de dades. Fals. Els models de sistema no poden assegurar propietats funcionals. Per a assegurar persistència de dades (és a dir, la seua durabilitat i tolerància a fallades) les dades haurien de replicar-se.
b	…és necessari per a comparar dos tipus de programació: asincrònica i multi-fil. Fals. Aquest no és l'objectiu del model de sistema explicat en el Tema 2.
c	…proporciona una bona base per a dissenyar algorismes distribuïts i per a raonar sobre la seua correcció abans d'iniciar el seu desenvolupament. Vertader. Aquest és l'objectiu principal dels models de sistema.
d	…demostra que les situacions de bloqueig d'activitats impedeixen que els serveis escalen. Fals. El model proposat no pot demostrar directament aquestes condicions. Proporciona la base per a analitzar algorismes, però els comportaments bloquejants depenen en molts casos d'alguns mecanismes utilitzats per a implantar els algorismes, no dels algorismes en si.

5. Aquesta és la millor solució per a proporcionar persistència de dades:

a	Usar servidors *stateful* (és a dir, servidors amb gestió d'estat en les seues interaccions). Fals. La gestió de l'estat d'una sessió en una interacció client-servidor no garanteix que l'estat mantingut en el domini servidor supere situacions de fallades. De fet, quan un servidor *stateful* falla, al client li resulta difícil reprendre la seua sessió amb un altre servidor que reemplace al que ha fallat.
b	Replicar les dades. Vertader. Les dades han de replicar-se per a superar situacions de fallades i garantir la seua durabilitat i capacitat d'accés.
c	Usar els discos durs més fiables. Fals. Per molt fiables que siguen, sempre podran desbaratar-se i perdre la seua informació.
d	Evitar els accessos concurrents a les dades. Fals. Els accessos concurrents poden posar en perill la consistència de les dades, però no la seua persistència.

6. El model de sistema senzill descrit en el Tema 2 està suportat directament per la programació asincrònica perquè…

a	…la programació asincrònica està basada en comunicació causal.

	Fals. La programació asincrònica no depèn de cap ordre en la comunicació.
b	…els processos en el model de sistema senzill són multi-fil. Fals. La programació asincrònica no implica processos multi-fil. De fet, els processos multi-fil solen ser sincrònics.
c	…hi ha una traducció directa entre guardes + accions en el model i esdeveniments + *callbacks* en la programació asincrònica. Vertader. En el model, les accions a executar pels processos estan protegides amb precondicions o guardes. En la programació asincrònica, un procés executa un fragment de codi (*callback*) quan es dóna un esdeveniment. Per tant, l'esdeveniment es comporta com la precondició i, una vegada es dóna, inicia l'execució del *callback* (que es correspon amb l'acció del model).
d	…els processos segueixen implícitament el model de fallades de parada en la programació asincrònica. Fals. Aquest tipus de programació no depèn de cap model de fallades.

7. Per a desenvolupar aplicacions escalables resulta més recomanable un sistema (middleware) de missatgeria (MoM, per les seues sigles en anglès) que la invocació d'objectes remots (RMI, en anglès) perquè…

a	MoM proporciona transparència d'ubicació, però RMI no pot proporcionar-la. Fals. RMI pot proporcionar transparència d'ubicació (p. ex., amb proxies) però MoM no sempre facilita aquest tipus de transparència. ZeroMQ és un exemple de MoM i els URL que utilitza no proporcionen transparència d'ubicació.
b	MoM és inherentment asincrònic, mentre RMI és sincrònic. Vertader. ZeroMQ és un exemple de MoM i proporciona comunicació asincrònica: un procés pot continuar amb la seua execució després de cridar a socket.send(), fins i tot en el patró de comunicacions REQ-REP. D'altra banda, RMI suspèn l'invocador fins que la resposta siga rebuda. Per tant, RMI és sincrònic i MoM asincrònic.
c	Els processos que utilitzen MoM assumeixen un espai compartit de recursos. En RMI els processos no poden compartir recursos. Fals. MoM proporciona una imatge sense compartició, a causa de la seua falta de transparència d'ubicació, mentre RMI proporciona normalment una imatge on tots els recursos es poden utilitzar (compartir) com a entitats locals (a causa de la seua transparència d'ubicació).
d	Els processos que utilitzen MoM estan replicats automàticament. En RMI, la replicació no està permesa. Fals. Tant MoM com RMI no estan directament relacionats amb la gestió de replicació. Ni un ni l'altre repliquen automàticament els agents o recursos (però tampoc ho impedeixen).

8. Els sistemes (middleware) de missatgeria persistent…

a	…proporcionen transparència d'ubicació. Fals. Ja s'ha comentat en la pregunta anterior que els MoM no proporcionen transparència d'ubicació.

b	…implanten la persistència automàticament en utilitzar un servei de noms. Fals. La persistència en la comunicació basada en missatges no depèn de cap servei extern de noms. La persistència depèn de la capacitat que tinguen els gestors de comunicació per a mantenir els missatges en trànsit quan el procés destinatari encara no estiga preparat per a rebre'ls.
c	…poden ser desenvolupats de manera senzilla utilitzant una implantació basada en gestors ("broker-based"). Vertader. Els gestors són els que han de mantenir els missatges per a proporcionar comunicació persistent.
d	…no poden ser utilitzats en comunicacions asincròniques. Fals. ZeroMQ és un exemple de MoM en el qual es combina comunicació asincrònica amb persistència parcial.

SEMINARIS

9. Considerant 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.');
```

Aquestes afirmacions són certes si s'assumeix que cap error avorta la seua execució:

a	Aquest programa mostra en totes les iteracions, entre altres informacions, el nom de l'últim fitxer facilitat en la línia d'ordres. Vertader. Aquest programa facilita un exemple de bucle en el qual cada iteració utilitza un *callback* asincrònic. En aquests casos, el *callback* no ha d'utilitzar el comptador d'iteracions del bucle. En ser asincrònic, el *callback* iniciarà la seua execució quan el bucle ja haja acabat d'iterar. Per això, normalment, aquests *callbacks* prendran el valor final del comptador d'iteracions. En aquest exemple, mostrarà el nom de l'últim fitxer en totes les iteracions.

b	Mostra el nom i grandària de cadascun dels fitxers rebuts en la línia d'ordres. Fals. Per a proporcionar aquesta eixida hauria d'utilitzar-se una clausura. Sense ella, les grandàries mostrades seran correctes, però els noms no, com ja s'ha descrit en l'apartat anterior.
c	Mostra "We have processed N files" al final de la seua execució, sent N el nombre de noms rebuts com a arguments. Fals. Encara que aquest missatge s'escriu en l'última línia del programa, aquesta línia formarà part del primer torn d'execució. Aquest torn finalitza abans que s'inicie l'execució del primer *callback*. Per això, serà el primer missatge mostrat en pantalla.
d	Descarta els dos primers noms de fitxer passats com a arguments en la línia d'ordres. Fals. No es descarta cap nom de fitxer passat com a argument. La línia 6 descarta la paraula "node" i el nom del programa, però no els arguments passats al programa.

10. Considerant el programa de la pregunta anterior…

a	Necessita múltiples torns (de la cua de planificació de l'intèrpret) per a completar la seua execució, ja què cada fitxer utilitza el seu propi torn. Vertader. Cada fitxer és llegit utilitzant un *callback* asincrònic. Cada *callback* s'executarà en un torn independent.
b	Genera una excepció i avorta si algun error ocorre quan intenta llegir un fitxer. Fals. Si ocorreguera algun error, es comunicarà en el primer argument del *callback*. El seu codi gestiona el primer paràmetre escrivint informació sobre l'error, però sense generar cap excepció ni avortar el procés.
c	El programa és incorrecte. Hauria d'utilitzar "var i=0" en inicialitzar la variable "i" per a ser correcte. Fals. El valor inicial d'aquesta variable és correcte. Els vectors s'indexen a partir de zero per omissió. Com el bucle do-while incremente el valor de "i" en la seua primera sentència, ha d'inicialitzar la variable a -1 per a accedir així a totes les components del vector "files".
d	Sempre mostra la mateixa grandària en totes les iteracions. Necessitaríem una clausura per a evitar aquest comportament defectuós. Fals. La grandària del fitxer es mostra correctament en totes les iteracions, perquè només depèn del paràmetre "data" i el seu valor es passa en cada invocació. La clausura es necessitaria per a passar adequadament el nom del fitxer.

11. En els algorismes d'exclusió mútua vistos en el Seminari 2 es pot afirmar que…

a	L'algorisme amb servidor central minimitza el nombre de missatges necessaris. Vertader. És el que necessita menys missatges d'entre els presentats en el Seminari 2. Per a fer això utilitza interaccions punt a punt i gestiona els accessos amb un coordinador. Així, n'hi ha prou amb un missatge per a sol·licitar l'entrada o comunicar l'eixida de la secció crítica i el permís per a entrar també es proporciona amb un sol missatge (la contestació a la sol·licitud). Altres algorismes empren difusions (N o N-1 missatges per difusió) per a fer el mateix o propaguen contínuament un token quan no hi ha cap sol·licitant actiu.

b	L'algorisme d'anell unidireccional té un retard de sincronització d'1 missatge. Fals. El seu retard de sincronització és d'1 missatge en el millor cas, però pot necessitar N-1 missatges en el pitjor cas.
c	El retard de sincronització de l'algorisme amb multidifusió i rellotges lògics és de 2N-2 missatges. Fals. El seu retard de sincronització és d'1 missatge.
d	La versió de l'algorisme de multidifusió i *quòrums* descrita en la presentació satisfà les 3 condicions de correcció per a aquests algorismes. Fals. És propens a interbloquejos i, a causa d'això, no pot complir la condició de vivacitat exigida a aquests algorismes.

12. Considerant aquest programa...

```
var ev = require('events');
var emitter = new ev.EventEmitter;
var num1 = 0;
var num2 = 0;
function emit_i1() { emitter.emit("i1") }
function emit_i2() { emitter.emit("i2") }
emitter.on("i1", function() {
   console.log( "Event i1 has happened " + ++num1 + " times.")});
emitter.on("i2", function() {
   console.log( "Event i2 has happened " + ++num2 + " times.")});
emitter.on("i1", function() {
   setTimeout( emit_i2, 3000 )});
emitter.on("i2", function() {
   setTimeout( emit_i2, 2000 )});
setTimeout( emit_i1, 2000 );
```

Aquestes afirmacions són certes:

a	L'esdeveniment “i1” ocorre només una vegada, 2 segons després de l'inici del procés. Vertader. L'última línia del programa estableix que l'esdeveniment **i1** ha d'emetre's en 2”. Aquest esdeveniment té dos *listeners*: un escriu quantes vegades s'ha donat **i1** i l'altre programa l'esdeveniment **i2** al cap de 3”. Cap dels *listeners* d'i2 generarà l'esdeveniment **i1**. Per tant, **i1** només ocorrerà una vegada, 2” després d'iniciar-se el procés.
b	L'esdeveniment “i2” no ocorre mai. Fals. L'esdeveniment **i2** succeeix múltiples vegades. La primera, 5” després de l'inici. Les restants de manera periòdica, amb intervals de 2”.
c	El període de “i2” és cinc segons. Fals. El seu període és 2”.
d	El període de “i1” és tres segons. Fals. Només ocorre una vegada.

13. En el programa de la pregunta anterior…

a	El primer esdeveniment “i2” ocorre cinc segons després d'haver-se iniciat el procés. Vertader. S'ha explicat en la pregunta anterior.

b	No es genera cap esdeveniment en la seua execució, perquè les crides a emit() són incorrectes. Fals. Els esdeveniments es generen correctament.
c	No es pot tenir més d'un *listener* per a cada esdeveniment. Per tant, el procés avorta generant una excepció en la tercera crida a emitter.on(). Fals. Els esdeveniments poden tenir múltiples *listeners*.
d	Cap dels seus esdeveniments ocorre periòdicament. Fals. L'esdeveniment **i2** succeeix cada dos segons.

14. El patró de comunicació REQ-REP de ØMQ es considera sincrònic perquè…

a	Segueix el patró d'interacció client/servidor i en aquest patró el client roman bloquejat fins que es reba una resposta. Fals. L'emissor de la petició pot continuar amb la seua execució. El socket REQ no el bloqueja. Les respostes a aquesta sol·licitud són gestionades mitjançant un *listener* per a l'esdeveniment **message**. Aquesta gestió és asincrònica.
b	Tant REQ com REP són *sockets* bidireccionals, és a dir, tots dos poden enviar i rebre missatges. Fals. Són bidireccionals, però la bidireccionalitat no implica sincronia (bloquejos).
c	La cua d'enviament del *socket* REQ té capacitat limitada. Només pot mantenir un missatge. Fals. Les cues d'enviament tenen major capacitat. No estan limitades a un sol missatge.
d	Els *sockets* REQ no poden transmetre una sol·licitud mentre la resposta anterior no s'haja rebut. Els *sockets* REP no poden enviar una resposta abans de la sol·licitud. Vertader. Aquesta és la característica que introdueix cert grau de sincronia en el patró de comunicació REQ-REP de ZeroMQ.

15. Considerant aquests dos programes…

```
// 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==2) client.end();
});
client.on('end', function() {
  console.log('client ' +
    'disconnected');
});
```

Les següents afirmacions són certes:

a	El servidor acaba després d'enviar la seua primera resposta al primer client. Fals. El codi del servidor no inclou cap instrucció que force la seua finalització (p. ex., un process.exit()). Quan una connexió es tanque, es mostrarà un missatge comunicant-ho, però no es fa res més. Per tant, el servidor gestionarà múltiples connexions, responent a cada missatge rebut en elles amb un altre missatge que concatenarà la paraula "World!" al contingut del missatge rebut.

b	El client no acaba mai. Vertader. No tanca mai la seua connexió oberta, ja que això ho faria després de processar el segon missatge rebut en ella, però el servidor només contesta una vegada. Per tant, aquest segon missatge no arribarà i el client no acabarà mai.
c	El servidor només pot gestionar una connexió. Fals. Els servidors que utilitzen el mòdul "net" poden gestionar múltiples connexions.
d	Aquest client no pot connectar-se al servidor. Fals. Tant client com servidor utilitzen el mateix nombre de port. Poden comunicar-se sense problemes si són executats en un mateix ordinador.

16. Els algorismes d'elecció de líder (vists en el Seminari 2)…

a	...no tenen condicions de seguretat. Fals. Per a ser correctes, tots els algorismes distribuïts han de respectar condicions de seguretat i vivacitat (almenys una condició de cada tipus). La condició de seguretat per als algorismes d'elecció de líder és que no ha d'haver-hi més d'un líder simultàniament en el sistema.
b	...poden estar indefinidament seleccionant un procés líder. Fals. La condició de vivacitat per a aquests algorismes exigeix que l'elecció s'efectue en algun moment. Per tant, no pot retardar-se indefinidament.
c	...han d'assegurar que un sol líder ha sigut triat. Vertader. Aquesta és la condició de seguretat.
d	...han de respectar l'ordre causal. Fals. Aquesta condició no s'exigeix a aquests algorismes. Va ser presentada com una condició de correcció per als algorismes d'exclusió mútua, però no per als d'elecció de líder.

17. Assumisca que es va a desenvolupar un servei d'exclusió mútua utilitzant NodeJS i ØMQ, emprant el primer algorisme explicat en el Seminari 2: el basat en un servidor central. La millor opció (d'entre les mostrades) per a fer això seria…

a	El servidor utilitzarà un *socket* DEALER i un *socket* ROUTER per a equilibrar la càrrega entre els seus clients. Fals. En aquest algorisme els clients utilitzen un patró petició-resposta per a implantar el protocol d'entrada a la secció crítica i un enviament unidireccional per a implantar el protocol d'eixida. El patró ROUTER-DEALER no està dissenyat per a gestionar aquests dos tipus d'interacció simultàniament. ROUTER-DEALER s'utilitza principalment perquè un *broker* propague i repartisca missatges entre un conjunt de servidors o treballadors. Aquest tipus d'interacció no s'utilitza en l'algorisme.

b	Cada client utilitzarà un *socket* DEALER per a interactuar amb el servidor. Vertader. Els sockets DEALER poden implantar tant el patró petició-resposta del protocol d'entrada com el d'enviament unidireccional del protocol d'eixida.
c	Cada client utilitzarà un *socket* REP per a interactuar amb el servidor. Fals. Els sockets REP no poden utilitzar-se per a iniciar un protocol d'entrada a la secció crítica. El socket REP no pot iniciar un enviament fins que un missatge de petició previ haja sigut rebut pel socket.
d	Cada client utilitzarà un *socket* SUB per a interactuar amb el servidor. Fals. Els sockets SUB només poden rebre missatges. No poden enviar-los. Per tant, no poden implantar cap dels dos protocols que hem explicat en la descripció del primer apartat.

18. Assumisca que es va a desenvolupar un servei d'exclusió mútua utilitzant NodeJS i ØMQ, emprant el 2n algorisme explicat en el Seminari 2: el d'anell (virtual) unidireccional. La millor opció (d'entre les mostrades) per a fer això seria…:

a	Utilitzar algun algorisme d'elecció de líder per a seleccionar un procés coordinador. Fals. L'algorisme d'anell és simètric. Tots els processos es comporten igual. Per tant, no es necessita cap coordinador en aquest algorisme.
b	Tots els processos tenen el mateix rol i necessiten un *socket* REP per a enviar missatges i un *socket* REQ per a rebre'ls. Fals. Tant REP com REQ impedeixen dos enviaments consecutius mentre no hi haja alguna recepció intermèdia. Amb això, no poden implantar un patró de comunicació unidireccional com el demanat en aquest algorisme.
c	Tots els processos tenen el mateix rol i necessiten un *socket* PUSH per a enviar el *token* i un *socket* PULL per a rebre'l. Vertader. Aquests sockets són unidireccionals i compleixen els requisits de comunicació d'aquest algorisme.
d	Tots els processos tenen el mateix rol i necessiten un *socket* PUB per a enviar el *token* i un *socket* DEALER per a rebre'l. Fals. Encara que un socket PUB puga emprar-se per a enviar missatges i un socket DEALER puga rebre'ls, el socket PUB difon allò que emet. Aquest tipus d'enviament no resulta apropiat per a implantar l'algorisme.

19. Considerant aquests programes...

```
//client.js
var zmq=require('zmq');
var rq=zmq.socket('dealer');
rq.connect('tcp://127.0.0.1:8888');
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('dealer');
rp.bindSync('tcp://127.0.0.1:8888');
rp.on('message', function(msg) {
  var j = parseInt(msg);
  rp.send([msg,(j*3).toString()]);
});
```

Les següents afirmacions són certes:

a	El client i el servidor intercanvien missatges sincrònicament, perquè tots dos segueixen un patró petició/resposta. Fals. Els dos utilitzen sockets DEALER i cap d'ells necessita bloquejar-se en cap acció

	relacionada amb missatges. De fet, el client pot arribar a enviar les seues 100 sol·licituds abans d'haver rebut la primera resposta.
b	El servidor retorna un missatge amb 2 segments al client. El segon segment conté un valor 3 vegades superior al del primer segment. Vertader. Aquesta és la funcionalitat del *listener* per a l'esdeveniment 'message' en el servidor.
c	El client i el servidor poden executar-se en diferents ordinadors. Interactuen sense problemes en aquest cas. Fals. Tots dos utilitzen 127.0.0.1 com a adreça IP en les seues crides a bindSync() o connect(). Això implica que els dos han d'executar-se en el mateix ordinador per a poder comunicar-se.
d	El client no envia cap missatge perquè la sentència `''+i` genera una excepció i el procés avorta en aquest punt. Fals. Aquesta sentència no genera cap error o excepció. El seu objectiu és tractar el valor de la variable **i** com una cadena.

20. Considere quina de les següents variacions generaria nous programes amb el mateix comportament observat en la pregunta 19 (A--> B significa que la sentència A és reemplaçada per la sentència B).

a	El *socket* 'rq' serà de tipus PULL i el *socket* 'rp' de tipus PUSH. Fals. El programa necessita comunicació bidireccional. PUSH i PULL són unidireccionals.
b	El *socket* 'rq' serà de tipus PUSH i el *socket* 'rp' de tipus PULL. Fals. El programa necessita comunicació bidireccional. PUSH i PULL són unidireccionals.
c	Client: rq.connect('tcp://127.0.0.1:8888'); --> rq.bindSync('tcp://*:8888'); Servidor: rp.bindSync('tcp://127.0.0.1:8888'); --> rp.connect('tcp://127.0.0.1:8888'); Vertader. Aquesta és la millor opció entre totes les presentades. El comportament serà idèntic al de la versió original si s'assumeix que només hi haurà un client i un servidor.
d	El *socket* 'rq' serà de tipus REP i el *socket* 'rp' de tipus REQ. Fals. Un socket REP no pot ser utilitzat per a iniciar la interacció amb un altre procés. Per tant, el client no pot usar un REP per a interactuar amb el servidor.