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.
b	El servei de correu electrònic és un exemple de sistema distribuït.
c	Els agents en un sistema distribuït no poden compartir cap recurs ja que estan situats en ordinadors diferents.
d	El programador d'una aplicació distribuïda no necessita preocupar-se sobre la tolerància a fallades perquè ja està garantida implícitament pel sistema distribuït.

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.
b	És un sistema LAMP, i tots els sistemes d'aquest tipus són escalables.
c	Utilitza interacció P2P i açò millora la seua escalabilitat.
d	Utilitza *proxies* inversos (com a "*caches*" o memòries cau) i replicació de components.

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.
b	Automatitzar la provisió d'infraestructura.
c	Proporcionar serveis distribuïts amb un model de pagament a mida ("pay-as-you-go").
d	Gestionar dades persistents amb un model de pagament a mida ("pay-as-you-go").

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

a	…garanteix persistència de dades.
b	…és necessari per a comparar dos tipus de programació: asincrònica i multi-fil.
c	…proporciona una bona base per a dissenyar algorismes distribuïts i per a raonar sobre la seua correcció abans d'iniciar el seu desenvolupament.
d	…demostra que les situacions de bloqueig d'activitats impedeixen que els serveis escalen.

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

a	Usar servidors *stateful* (és a dir, servidors que gestiones estat en les seues interaccions).
b	Replicar les dades.
c	Usar els discos durs més fiables.
d	Evitar els accessos concurrents a les dades.

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.
b	…els processos en el model de sistema senzill són multi-fil.
c	…hi ha una traducció directa entre guardes + accions en el model i esdeveniments + *callbacks* en la programació asincrònica.
d	…els processos segueixen implícitament el model de fallades de parada en la programació asincrònica.

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

a	MoM proporciona transparència d'ubicació, però RMI no pot proporcionar-la.
b	MoM és inherentment asincrònic, mentre RMI és sincrònic.
c	Els processos que utilitzen MoM assumeixen un espai compartit de recursos. En RMI els processos no poden compartir recursos.
d	Els processos que utilitzen MoM estan replicats automàticament. En RMI, la replicació no està permesa.

8. Els sistemes (middleware) de missatgeria persistent…

a	…proporcionen transparència d'ubicació.
b	…implanten la persistència automàticament en utilitzar un servei de noms.
c	…poden ser desenvolupats de manera senzilla utilitzant una implantació basada en gestors (“broker-based”).
d	…no poden ser utilitzats en comunicacions asincròniques.

TSR

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 una altra informació, el nom de l'últim fitxer facilitat en la línia d'ordres.
b	Mostra el nom i grandària de cadascun dels fitxers rebuts en la línia d'ordres.
c	Mostra "We have processed N files" al final de la seua execució, sent N el nombre de noms rebuts com a arguments.
d	Descarta els dos primers noms de fitxer passats com a arguments en la línia d'ordres.

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 que cada fitxer utilitza el seu propi torn.
b	Genera una excepció i avorta si algun error ocorre quan intenta llegir un fitxer.
c	El programa és incorrecte. Hauria d'utilitzar "var i=0" en inicialitzar la variable "i" per a ser correcte.
d	Sempre mostra la mateixa grandària en totes les iteracions. Necessitaríem una clausura per a evitar aquest comportament defectuós.

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.
b	L'algorisme d'anell unidireccional té un retard de sincronització d'1 missatge.
c	El retard de sincronització de l'algorisme amb multidifusió i rellotges lògics és de 2N-2 missatges.
d	La versió de l'algorisme de multidifusió i quòrums descrita en la presentació satisfà les 3 condicions de correcció per 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.
b	L'esdeveniment “i2” no ocorre mai.
c	El període de l’esdeveniment “i2” és cinc segons.
d	El període de l’esdeveniment “i1” és tres segons.

13. En el programa de la pregunta anterior…

a	El primer esdeveniment “i2” ocorre cinc segons després d'haver-se iniciat el procés.
b	No es genera cap esdeveniment en la seua execució perquè les crides a emit() són incorrectes.
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().
d	Cap dels seus esdeveniments ocorre periòdicament.

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.
b	Tant REQ com REP són *sockets* bidireccionals, és a dir, tots dos poden enviar i rebre missatges.
c	La cua d'enviament del *socket* REQ té capacitat limitada. Només pot mantenir un 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.

TSR

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.
b	El client no acaba mai.
c	El servidor només pot gestionar una connexió.
d	Aquest client no pot connectar-se al servidor.

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

a	...no tenen condicions de seguretat.
b	...poden estar indefinidament seleccionant un procés líder.
c	...han d'assegurar que un únic líder ha sigut triat.
d	...han de respectar l'ordre causal.

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) para a fer això seria…

a	El servidor utilitzarà un *socket* DEALER i un *socket* ROUTER per a equilibrar la càrrega entre els seus clients.
b	Cada client utilitzarà un *socket* DEALER per a interactuar amb el servidor.
c	Cada client utilitzarà un *socket* REP per a interactuar amb el servidor.
d	Cada client utilitzarà un *socket* SUB per a interactuar amb el servidor.

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

a	Utilitzar algun algorisme d'elecció de líder per a seleccionar un procés coordinador.
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.
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.
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.

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, ja què tots dos segueixen un patró petició/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.
c	El client i el servidor poden executar-se en diferents ordinadors. Interactuen sense problemes en aquest cas.
d	El client no envia cap missatge perquè la sentencia `''+i` genera una excepció i el procés avorta en aquest punt.

20. Considere quin 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.
b	El *socket* 'rq' serà de tipus PUSH i el *socket* 'rp' de tipus PULL.
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');
d	El *socket* 'rq' serà de tipus REP i el *socket* 'rp' de tipus REQ.