TSR

Aquest examen inclou 20 qüestions d'opció múltiple. Cadascuna d'elles solament 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.
b	Proporcionar transparència de fallades.
c	Permetre la compartició de recursos, especialment d'aquells dispositius que resulten cars i puguen accedir-se de manera remota.
d	Facilitar el desplegament de les aplicacions.

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

a	SLA.
b	SaaS.
c	IaaS.
d	*Middleware* de comunicacions (basat en missatges).

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.
b	Les seues interaccions client-servidor han de basar-se en un *middleware* de comunicacions asincròniques basat en missatges.
c	Proporciona serveis de programari distribuïts als seus clients, generalment en un model de pagament per ús.
d	Els usuaris dels serveis decideixen de quina manera es desplegaran els programes.

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

a	…aconsegueix que el desplegament d'aplicacions siga trivial.
b	…està basat en esdeveniments i assegura l'execució atòmica de cada acció.
c	…proporciona transparència de fallades.
d	…utilitza *proxies* inversos i, a causa d'això, és altament escalable.

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

a	Els rellotges lògics generen processos sincrònics.
b	L'ordre (sincrònic) dels missatges està basat a limitar el temps de propagació dels missatges.
c	Els processos sincrònics avancen en passos. En cada pas, tot procés completa una acció.
d	La comunicació sincrònica requereix que els canals mantinguen els missatges enviats fins que els receptors puguen acceptar-los.

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.
b	L'ús d'estàndards proporciona una interfície d'alt nivell en els *middleware*. Així, les tasques de programació resulten senzilles.
c	Les APIs proporcionades per sistemes *middleware* no sempre són estàndard. ZeroMQ n'és un exemple.
d	Els sistemes *middleware* no han de respectar cap estàndard, ja que els estàndards solament es defineixen per als elements interns dels sistemes operatius.

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

a	El protocol IP.
b	Un servei de noms distribuït.
c	TCP.
d	El servidor *Apache*.

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.
b	No proporcionen transparència d'ubicació.
c	Faciliten un baix nivell d'abstracció, complicant els programes resultants.
d	El seu comportament és excessivament asincrònic, i per això no poden depurar-se fàcilment.

TSR

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);
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.
b	Mostra el nom i grandària de cada fitxer rebut com a argument.
c	Mostra "We have processed 0 files" com el seu primer missatge en pantalla.
d	Descarta alguns dels noms de fitxer proporcionats com a arguments després dels elements "node nom-programa".

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

a	Necessita diversos torns per a completar la seua execució perquè cada fitxer a llegir necessita un torn per al seu *callback*.
b	L'increment de la "i" (instrucció "i++") està situat incorrectament. Hauria d'estar dins del *callback*.
c	Aquest programa mostra un error i finalitza si s'han passat menys de cinc noms de fitxer com a arguments.
d	Mostra la mateixa grandària en totes les iteracions. Es necessita una clausura per a evitar aquest comportament incorrecte.

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.
b	L'algorisme d'anell virtual unidireccional no perd el *token* si el procés actualment en la secció crítica falla.
c	L'algorisme de difusió amb rellotges lògics usa menys missatges que l'algorisme de difusió basat en quòrums.
d	L'algorisme de difusió amb rellotges lògics compleix les tres condicions de correcció del problema d'exclusió mútua.

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.
b	L'esdeveniment "i2" no ocorre mai.
c	L'esdeveniment "i2" ocorre periòdicament, cada tres segons.
d	L'esdeveniment "i1" ocorre periòdicament, cada dos segons.

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.
b	Com tots dos esdeveniments utilitzen el mateix *listener*, tots dos mostren missatges amb exactament el mateix contingut quan ocorren.
c	El primer esdeveniment "i2" ocorre dos segons després del primer esdeveniment "i1".
d	Cap dels esdeveniments ocorre dues o més vegades.

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

a	Tots dos *sockets* estan connectats o han realitzat un "bind()" sobre el mateix URL.
b	Tots dos *sockets* són bidireccionals.
c	El *socket* REP utilitza una operació sincrònica per a manejar els missatges rebuts.
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).

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.
b	El client no acaba mai.
c	El servidor pot gestionar múltiples connexions.
d	El client no pot connectar amb el servidor.

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

a	...necessiten l'execució prèvia d'un algorisme d'exclusió mútua perquè la identitat del líder es guarda en un recurs compartit i solament pot modificar-la un procés.
b	…necessiten consens entre tots els processos participants: tots han de triar un mateix líder.
c	…no necessiten identitats úniques per a cada procés.
d	...han de respectar ordre causal.

17. Es vol escriure un programa d'elecció de líder en NodeJS i ØMQ, usant el primer algorisme del Seminari 2: el d'anell virtual. Per a fer això, la millor de les següents opcions és:

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.
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.
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.
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.

18. Es vol escriure un programa d'elecció de líder en NodeJS i ØMQ, utilitzant el 2n algorisme del Seminari 2: l'algorisme intimidador ("*bully*"). 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".
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.
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.
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.

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

a	El tipus PUB, per a gestionar les seues difusions.
b	El tipus PUSH, per a gestionar múltiples operacions send() asincròniques.
c	El tipus REQ, en cas d'estar connectat a múltiples *sockets* REP.
d	El tipus ROUTER, utilitzant una cua d'enviament per a cada connexió.

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".
b	Cada servidor rep, gestiona i contesta les 100 sol·licituds. Així, el client rep i mostra 200 respostes.
c	Algunes sol·licituds inicials es perden perquè el client ha sigut iniciat abans que començara el primer servidor.
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.