Este examen contiene 20 cuestiones de opción múltiple. En cada una de ellas solo una de sus respuestas es correcta. Las contestaciones deben presentarse en una hoja entregada aparte. Las respuestas correctas aportan 0.5 puntos a la nota del parcial mientras que las incorrectas restan una décima.

En la hoja de respuestas debes rellenar la casilla elegida cuidadosamente. Utiliza un lápiz o un bolígrafo (negro o azul oscuro). Se puede usar "Tipp-Ex" o algún corrector similar. En ese caso, NO INTENTES DIBUJAR DE NUEVO LA CASILLA QUE HAYAS BORRADO.

TEORÍA

1. Los sistemas distribuidos...

A	...están formados generalmente por múltiples agentes que se ejecutan concurrentemente. Esos agentes mantienen cierto estado independiente.
B	...no necesitan ningún mecanismo de comunicación entre ordenadores.
C	...siempre utilizan interacciones cliente-servidor.
D	...nunca tendrán condiciones de carrera.
E	Todas las anteriores.
F	Ninguna de las anteriores.

2. La computación en la nube (*Cloud computing*)...

A	...es uno de los paradigmas actuales de prestación de servicios en la computación distribuida.
B	...tiene como objetivo la prestación de servicios de cómputo de una manera escalable y eficiente.
C	...sigue un modelo de "pago por uso".
D	...usa generalmente infraestructuras virtualizadas.
E	Todas las anteriores.
F	Ninguna de las anteriores.

3. En un sistema distribuido, la interacción entre sus agentes...

A	…nunca debe llevarse a cabo. Si interactuaran, el sistema sería concurrente en lugar de distribuido.
B	…se realiza intercambiando mensajes o compartiendo memoria.
C	…se realiza sin compartir memoria. La compartición de memoria está prohibida en los sistemas distribuidos.
D	…se consigue cuando todos los agentes residan en un mismo ordenador.
E	Todas las anteriores.
F	Ninguna de las anteriores.

4. El modelo de programación guarda / acción ...

A	Se sigue en la programación multi-hilo, donde las secciones críticas equivalen a las guardas y los hilos (“threads”) equivalen a las acciones.
B	Se sigue en la programación asincrónica (o dirigida por eventos), donde los eventos equivalen a las acciones y las funciones “callback” de los eventos equivalen a las guardas.
C	Se sigue en la programación multi-hilo, donde los hilos (“threads”) equivalen a guardas que se activan y se suspenden, y las operaciones en las secciones críticas equivalen a las acciones.
D	Se sigue en la programación asincrónica (o dirigida por eventos), donde los eventos equivalen a las guardas y las funciones “callback” de los eventos equivalen a las acciones.
E	Todas las anteriores.
F	Ninguna de las anteriores.

5. Algunas características relevantes de los modelos de sistemas distribuidos son...

A	Se centran en las principales propiedades del comportamiento del sistema.
B	Facilitan una buena herramienta para razonar sobre la corrección de los algoritmos y protocolos basados en ellos.
C	Su alto nivel de abstracción.
D	Facilitan una base para discutir sobre la imposibilidad de resolver problemas en ciertos sistemas distribuidos (p.ej., el consenso en sistemas asincrónicos).
E	Todas las anteriores.
F	Ninguna de las anteriores.

6. Los elementos a considerar en un modelo de sistema pueden ser...

A	La arquitectura del equipo, el sistema operativo, el middleware y el lenguaje de programación.
B	Procesos, eventos, aspectos de comunicación, fallos, gestión del tiempo y nivel de sincronía.
C	Nivel físico, nivel de enlace, nivel de red, nivel de transporte y nivel de aplicación.
D	Sistema gestor de bases de datos, middleware e interfaz de usuario.
E	Todas las anteriores.
F	Ninguna de las anteriores.

7. En la programación de sistemas distribuidos, el uso del middleware es aconsejable porque...

A	Introduce múltiples transparencias, ocultando detalles de bajo nivel y ofreciendo una interfaz uniforme.
B	Tiene una implantación sencilla, y poca complejidad en los elementos manejados.
C	Proporciona una operativa estandarizada, comprensible y bien definida.
D	Facilita la interoperabilidad, la interacción con productos de terceras partes.
E	Todas las anteriores.
F	Ninguna de las anteriores.

8. Los problemas que encontramos en los sistemas distribuidos orientados a objetos son:

A	Todos los objetos parecen ser locales y esto puede generar largos intervalos para completar su invocación en caso de que sean remotos.
B	Los objetos mantienen estado y ese estado se compartirá entre los agentes que invoquen sus métodos. Esto puede provocar problemas de consistencia.
C	Su estado compartido necesita mecanismos de control de concurrencia. Esto puede ocasionar bloqueos, evitando que los sistemas sean escalables.
D	Sus mecanismos de invocación facilitan una alta transparencia de ubicación. Esto exige protocolos de recuperación complejos para gestionar los fallos.
E	Todas las anteriores.
F	Ninguna de las anteriores.

SEMINARIOS

9. Considérese el siguiente programa (incompleto) escrito en Node:

```
function logaritmo(x,b) { return Math.log(x)/Math.log(b) }
function logBase ... // a completar
log2 = logBase(2);
log8 = logBase(8);
console.log("Logarithm base 2 of 1024 = " + log2(1024));
console.log("Logarithm base 8 of 4096 = " + log8(4096));
```

¿Cuál implementación de la función `logBase` sería correcta?

A	`function logBase(b) { return logaritmo(x,b) }`

B	`function logBase(b) {`
	`  return function(x) { return logaritmo(x,b) }`
	`}`

C	`function logBase(x) {`
	`  return function(b) { logaritmo(x,b) }`
	`}`

D	`function logBase(x) {`
	`  return function(b) { return logaritmo(x,b) }`
	`}`

E	Todas las anteriores.
F	Ninguna de las anteriores.

10. Considérese el siguiente programa escrito en Node:

```
var fruits = ["Banana", "Orange", "Lemon", "Apple"];
var numbers = [7, 3, "Cloud", 9];
var funcs = [function(x) {return 2*numbers[x]},
             function(x) {return fruits[x]}];
var s = "";
for (var i=0; i<2; i++)
  for (var j=0; j<5; j=j+2)
     s += funcs[i](j) + ", ";
console.log(s);
```

Al ejecutarlo, la salida que se mostrará en consola será:

A	14, 6, NaN, Banana, Orange, Lemon,
B	14, NaN, NaN, Banana, Lemon, undefined,
C	14, 2Cloud, undefined, Banana, Lemon, undefined,
D	Banana, 14, Lemon, NaN, undefined,
E	No se mostraría nada, salvo un mensaje de error indicando que el array `numbers` está mal definido, por contener valores de diferentes tipos.
F	Ninguna de las anteriores.

11. **Considérese la siguiente función escrita en Node:**

```
function f(x,y) {
   x = x || 'naranja'; y = y || 98;
   console.log('x='+x+' y='+y);
}
```

Indique cuál sería la salida que se mostrará en consola si se ejecuta:

```
f(36);          f(undefined,'manzana');    f(45,0,67);
```

A	x=36 y=98	x=undefined y=manzana	x=45 y=0
B	x=36 y=98	x=naranja y=manzana	x=45 y=0
C	x=36 y=98	x=naranja y=manzana	x=45 y=98
D	x=36 y=36	x=naranja y=manzana	x=45 y=67
E	No se mostraría nada, salvo mensajes de error pues hay invocaciones incorrectas (por su número de argumentos) de la función f.		
F	Ninguna de las anteriores.		

12. **Considérese el siguiente programa escrito en Node:**

```
var eve = new (require('events')).EventEmitter;
var s = "print";
var n = 0;
var handler = setInterval( function(){eve.emit(s);}, 1000 );
eve.on(s, function() {
   if ( n < 2 ) console.log("Event", s, ++n, "times.");
   else clearInterval(handler);
});
```

Si se ejecuta este programa indique, en relación a la salida que se mostrará en consola y al tiempo de ejecución, cuál de las siguientes opciones es la correcta:

A	Event print 1 times. Event print 2 times.	Y concluiría después de 3 segundos.
B	Event print 1 times. Event print 2 times. Event print 3 times.	Y concluiría después de 4 segundos.
C	Event print 1 times. Event print 2 times. …	Y no concluiría. Cada segundo, mostraría una nueva línea con el número incrementado en una unidad.
D	Event print 0 times. Event print 1 times. …	Y no concluiría. Cada 10 segundos, mostraría una nueva línea con el número incrementado en una unidad.
E	No se mostraría nada, porque no está bien definido el objeto listener.	Y no concluiría, pues se emite cíclicamente el evento “print”.
F	Ninguna de las anteriores.	

13. Considerando el programa siguiente...

```
var http = require('http');
var fs = require('fs');
http.createServer(function(request,response) {
  fs.readdir(__dirname, function(err,data) {
    if (err) {
      response.writeHead(404, {'Content-Type':'text/plain'});
      response.end('Unable to read directory '+__dirname);
    } else {
      response.writeHead(200, {'Content-Type':'text/plain'});
      response.write('Directory: ' + __dirname + '\n');
      response.end(data.toString());
    }
  })
}).listen('1337');
```

Seleccione las opciones correctas:

A	Este programa genera una excepción y aborta en caso de no poder leer el contenido del directorio actual.
B	Este programa es un servidor web que responde con el nombre y lista de ficheros en el directorio actual.
C	Este programa no funciona porque no ha declarado la variable "__dirname" y no ha importado el módulo 'process' donde está definida.
D	Este programa no funciona porque 'data' es un vector de nombres de fichero y los vectores no pueden ser transformados en cadenas.
E	Todas las anteriores.
F	Ninguna de las anteriores.

14. Algunos problemas del algoritmo de exclusión mutua con servidor central son...

A	No cumple su condición de vivacidad.
B	No cumple su condición de seguridad.
C	Necesita más mensajes que los demás algoritmos vistos en el Seminario 2 para resolver el problema de exclusión mutua.
D	Es frágil en situaciones de fallo. El servidor central es un punto único de fallo.
E	Todas las anteriores.
F	Ninguna de las anteriores.

15. Los algoritmos de elección de líder...

A	...son un subconjunto de los algoritmos de consenso.
B	...necesitan que todos los procesos tengan un identificador distinto.
C	...usan un criterio determinista para elegir al líder.
D	...exigen que se elija solo a un proceso.
E	Todas las anteriores.
F	Ninguna de las anteriores.

16. Supongamos que se necesita implantar un servicio de chat utilizando node.js y ØMQ. El servidor difunde los mensajes de los usuarios y nunca debe suspenderse tratando de enviar un mensaje (de cualquier tipo). Los programas clientes envían los mensajes de los usuarios al servidor, esperan los mensajes reenviados por el servidor e informan al servidor cuando un usuario se incorpora o abandona el sistema. Para implantar este servicio de chat...

A	El servidor debe usar un socket PULL y otro REP para interactuar con los clientes.
B	El servidor debe usar un socket SUB y otro REQ para interactuar con los clientes.
C	El servidor debe usar un socket PUB y otro PULL para interactuar con los clientes.
D	El servidor debe usar un socket REP y otro SUB para interactuar con los clientes.
E	Todas las anteriores.
F	Ninguna de las anteriores.

17. Supongamos que hemos implantado un servicio soportado por múltiples (p.ej., 10) procesos servidores ubicados en ordenadores diferentes. Esos servidores utilizan sockets REP y sus clientes usan sockets REQ. Si construimos un broker con un socket ROUTER como *front-end* y un socket DEALER como *back-end* (y para ambos se realiza un bind()), entonces...

A	Los clientes no necesitan conocer cuántos procesos servidores hay.
B	Los clientes no necesitan conocer las direcciones y puertos de cada proceso servidor.
C	La cantidad de procesos servidores puede variar dinámicamente. Ellos deben conectarse al socket *back-end* para que el broker pueda utilizarlos.
D	El broker no debe modificar ningún segmento de los mensajes para propagarlos del *front-end* al *back-end* y del *back-end* al *front-end*.
E	Todas las anteriores.
F	Ninguna de las anteriores.

Para contestar a las siguientes 2 cuestiones (nº 18 y 19), considérense los siguientes programas Node con ØMQ. Un servidor (*server.js*):

```
var zmq = require('zmq')
var rep = zmq.socket('rep')
rep.bindSync('tcp://127.0.0.1:'+process.argv[2])
var n = 0
rep.on('message', function(msg) {
  console.log('Request: ' + msg)
  rep.send('World ' + ++n)
})
```

Y un cliente (*client.js*):

```
var zmq = require('zmq')
var req = zmq.socket('req')
req.connect('tcp://127.0.0.1:'+process.argv[2])
req.connect('tcp://127.0.0.1:'+process.argv[3])
var n = 0
setInterval( function() { req.send('Hello ' + ++n) }, 100 )
req.on('message', function(msg) {
  console.log('Response: ' + msg)
})
```

18. Considérense los anteriores programas Node con ØMQ (*server.js* y *client.js*). Si, en 3 terminales, se ejecutaran 2 servidores y 1 cliente mediante:

```
node server 8001
node server 8002
node client 8001 8002
```

Las primeras líneas que se mostrarán en las terminales de los servidores serán:

A	En una terminal:
	`Request: Hello 1`
	`Request: Hello 3`
		Y en la otra terminal:
		`Request: Hello 2`
		`Request: Hello 4`

B	En ambas terminales:
	`Request: Hello 1`
	`Request: Hello 2`	

C	En ambas terminales (siendo x, y, z… números tales que x < y < z < …):
	`Request: Hello x`
	`Request: Hello y`
	`Request: Hello z`
	`...`	

D	En una terminal:
	`Request: Hello 1`
	`Request: Hello 2`
		Y en la otra terminal:
		`Request: Hello 3`
		`Request: Hello 4`

E	No se mostraría nada, dado que el cliente no sabría a cuál de los servidores enviar sus peticiones. (Para un funcionamiento correcto, el cliente debería conectarse a un socket ROUTER).	
F	Ninguna de las anteriores.	

19. Considérense los mismos programas, y el mismo escenario de ejecución, de la cuestión anterior. Las primeras líneas que se mostrarán en la terminal del cliente serán:

A	`Response: World 1`
	`Response: World 2`
	`Response: World 3`
	`Response: World 4`

B	`Response: World 1`
	`Response: World 1`
	`Response: World 2`
	`Response: World 2`

C	`Response: World 1`
	`Response: World 3`
	`Response: World 5`
	`Response: World 7`

D	`Response: World 1`
	`Response: World 3`
	`Response: World 2`
	`Response: World 4`

E	No se mostraría nada, dado que, como el cliente no sabría a cuál de los servidores enviar sus peticiones, ninguno de los servidores podría enviar respuestas.
F	Ninguna de las anteriores.

20. Considérense los siguientes programas Node con ØMQ. Un publicador:

```
var zmq = require('zmq')
var pub = zmq.socket('pub').bindSync('tcp://*:5555')
var count = 0
setInterval(function() {
  pub.send('PRG ' + count++)
  pub.send('TSR ' + count++)
}, 1000)
```

Y un suscriptor:

```
var zmq = require('zmq')
var sub = zmq.socket('sub')
sub.connect('tcp://localhost:5555')
sub.subscribe('TSR')
sub.on('message', function(msg) {
  console.log('Received: ' + msg)
})
```

Si se ejecutara, en primer lugar, el publicador y, tres segundos después, el suscriptor. Las primeras líneas de la salida que se mostrarán en la terminal del suscriptor serán:

A	Received: TSR 1 Received: TSR 2 Received: TSR 3 …
B	Received: TSR 1 Received: TSR 3 Received: TSR 5 …
C	Received: PRG 4 Received: TSR 5 Received: PRG 6 …
D	Received: TSR 5 Received: TSR 7 Received: TSR 9 …
E	Received: PRG 0 Received: TSR 1 Received: PRG 2 …
F	Ninguna de las anteriores.