TSR

This exam consists of 20 multiple choice questions. In every case only one answer is correct. You should answer in a separate sheet. If correctly answered, they contribute 0.5 points to the exam grade. If incorrectly answered, the contribution is negative: -0.167. So, think carefully your answers.

THEORY

1. This is not one of the "main reasons" for using distributed systems:

a	To improve the efficiency of their applications, splitting the problem to be solved in multiple pieces, and running multiple agents in different computers to solve it. When an application is distributed, the data to be processed may be divided in multiple portions and spread among agents. In this way, its efficiency may be improved. Therefore, this is a reason for distributing an application.
b	To achieve failure transparency. When an application is run by a single agent, if that agent fails, then the entire application has failed. However, in a distributed application, multiple agents collaborate. Thus, when one fails, the others may still complete the intended execution providing a result. Thus, distributed applications can achieve failure transparency and this is one of their main advantages.
c	To allow resource sharing, especially when those devices are expensive devices that may be remotely accessed. That kind of sharing cannot happen in systems composed by a single computer. Thus, that is one of the main reasons for using distributed systems.
[✓ d]	To make application deployment as easy as possible. Distributed applications consist of multiple components that are usually installed onto several computers. This complicates their configuration and their dependency resolution. Once they are running, users expect a continuous service provision (i.e., high availability). Such service continuity is also difficult to achieve in case of a software upgrade. Therefore, when distributed applications are compared with stand-alone applications, their deployment isn't easy.

2. In cloud systems, hardware virtualisation technology is a basic mechanism for this service model:

a	SLA. These initials correspond to "Service Level Agreement". SLAs are not an example of service model. They are an agreement between a service provider and a service customer on the quality of service level to be guaranteed.

b	SaaS. Hardware virtualisation technologies are mainly used by infrastructure providers. The SaaS model is focused on the provision of software services. Therefore, it is not mainly centred in the hardware infrastructure to be used.
[✓ c]	IaaS. Hardware virtualisation technologies are mainly used by infrastructure providers. Since the IaaS model is mainly focused on infrastructure provision, virtualisation is a key technology in that model.
d	Message-based middleware. Middleware layers are not examples of cloud service models.

3. In the SaaS model in cloud systems, these statements are true:

a	Its services may be locally accessed, without any network intervention, using hardware virtualisation. The SaaS model does not manage “local” service accesses. Cloud service models are intended for remote service provision in all cases. Therefore, the first half of this statement doesn’t make sense. Moreover, hardware virtualisation is not the key part for this model.
b	Its client-server interactions must be based on an asynchronous message-oriented middleware. Asynchronous message-oriented interactions are not mandatory in that model.
[✓ c]	It provides distributed software services to its customers, in some cases under a pay-as-you-go model. This statement provides a concise definition of that model.
d	Service users decide how the software must be deployed. Software deployment is a responsibility of the service providers. Service users do not participate in those decisions or tasks.

4. Unit 2 recommends the asynchronous programming paradigm because that paradigm…

a	…makes application deployment trivial. Programming paradigms or models have almost no effect on application deployment. Software deployment encompasses all stages in the software life cycle once that software has been developed (installation, configuration, activation, upgrading, scaling, deactivation, removal...) and many of those stages do not depend on the programming paradigm that has been used in the development stage.

b	…is event-based and ensures the atomic execution of each action. True. Both characteristics are respected in that programming paradigm. Moreover, those characteristics are also convenient, since atomic actions prevent race conditions from happening, and event orientation provides a direct matching to the actual behaviour of distributed actions.
c	…provides failure transparency. Failure transparency depends on the usage of agent redundancy. The usage of redundancy (or replication) must be considered in the software design stage and its degree (amount of replicas per agent) may be varied at run-time depending on several factors (e.g., the current workload). The programming paradigm is not relevant in that regard.
d	…is based on reverse proxies and, because of this, is highly scalable. Reverse proxies are an example of forwarding and caching mechanism. They are not directly related with the asynchronous programming paradigm.

5. Regarding the aspects of synchrony described in Unit 2, it is true that:

a	Logical clocks ensure process synchronisation. Process synchronisation is related to advancing in synchronous steps. Logical clocks do not impose that kind of advancing. They tag communication events, and they could provide a basis for achieving that synchronisation, but such level of synchronisation must be ensured using other complementary mechanisms. Indeed, what is needed is a careful algorithm design.
b	Synchronous message order is based on setting bounds on message propagation time. If there are bounds on message propagation time, then we say that we have "synchronous channels". "Synchronous message order" implies that messages being transmitted will respect a given delivery order; e.g., FIFO, causal, total, etc. To this end, nodes need reception buffers (where received messages are kept until they may be delivered to their target agent) and they also use a protocol for ensuring that intended order.
c	Synchronous processes progress in steps. At each step, every process completes an action. This is a concise and valid definition of "synchronous process".
d	Synchronous communication requires that channels keep sent messages until the receiver is ready for accepting them. That is the definition of "persistent communication". "Synchronous communication" is one that blocks the sender agent until the receiver acknowledges the correct delivery of a given message. Those two types of communication consider different aspects. Therefore, that statement is false.

6. These sentences relate middleware and standards. What is false?

a	The usage of standards facilitates that middleware and agent implementations from different companies interoperate. That is true. Standards facilitate interoperability.

b	The usage of standards provides a high-level interface in middleware layers. With this, programming tasks become easier. True. Many standards internally use low-level protocols, providing a high-level interface that is more comfortable for programmers.
c	Middleware APIs may be non-standard. ZeroMQ is an example of this kind. True. ZeroMQ is an example of middleware for asynchronous communication. There is a standard in that same field: AMQP, but ZeroMQ does not comply with that standard.
d	Middleware layers cannot comply with any standard, since standards are only defined for internal elements of the operating system kernel. False. Middleware layers usually comply with some standards. Mainly, when interoperability is one of their goals. Standards are not limited to the internals of operating systems.

7. Which of these communication-related elements can be considered an example of middleware?

a	The IP protocol. Middleware is a software layer placed on top of the operating system and below the applications that use it. The IP protocol is at the network layer (layer 3) in a communications-related layered architecture. Layers 2 to 4 are usually managed by the operating system. Therefore, that protocol is below the middleware.
b	A distributed naming service. This kind of service is placed above the operating system and provides a base for achieving location transparency in a distributed system. Its services are used by other applications that are conceptually placed above it in a dependency-based layered architecture. Therefore, it can be considered an example of middleware.
c	TCP. TCP is a transport protocol (layer 4). It is managed by the operating system. Therefore, it is below any middleware.
d	The Apache server. The Apache server is an example of "web server". Thus, it may be considered a top-layer "application". It is not a middleware providing some kind of distribution transparency.

8. In the context of middleware examples, which are the problems of distributed object systems when they are compared with messaging systems?

a	Their potential high coupling that could lead to a blocking behaviour when shared resources are concurrently used by many agents. True. Object systems provide location transparency. All object invocations seem to be local, but they may access remote objects. This means that there may be multiple concurrent invocations on the same object, requiring some concurrency control mechanisms. Those mechanisms may block agent activity for long intervals (being their length proportional to the degree of concurrency). In the end, this may reduce the efficiency and scalability of those systems.

b	They are not location transparent. False. The interactions use proxies and skeletons and these elements provide location transparency.
c	They provide a low level of abstraction, complicating the resulting programs. Regarding the level of abstraction, both object-oriented programming languages and object-based system are high-level. Applications are easy to develop in those frameworks.
d	Their behaviour is highly asynchronous, being almost impossible to debug. The level of asynchrony in the behaviour of a distributed application is not directly related with the programming paradigm. Indeed, object-oriented programs use client-server interactions and those interactions are synchronous, since once the client has sent a request, it remains blocked waiting for a reply.

SEMINARS

9. Considering this program:

```
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.');
```

These sentences are true if we assume that no error aborts this program execution and sufficient file names have been given as arguments:

a	Due to the readFile() callback asynchrony, this program is unable to show in each iteration the intended file name and size. It is able to show the correct length for each file, but not their names. The length is computed using one of the callback arguments (“data”). Thus, it is computed without any trouble. However, the name is taken from the “files” array, using the current value of the “i” variable. Since readFile() is asynchronous, the callbacks will be called once the do..while loop has terminated all its iterations. At that time, the value of “i” equals “file.length” and the “files[file.length]” element of that array has an “undefined” value.

b	It prints the name and length for each one of the files passed as command-line arguments. It is unable to print the names, as it has been already explained in part "a".
c	It prints "We have processed 0 files" as its first message. Although that is the first message to be printed, it doesn't show "0" files but the correct amount of files passed from the command line. Note that "files.length" is already defined at that time and its value must exceed 2 (since the second program line requires that process.argv.length exceeds 4).
d	It discards some file names given as arguments to this program, after the "node program-name" elements. No. The statement "process.argv.slice(2)" returns a copy of the process.argv array without its first two elements. Those elements hold the "node" and "program-name" words. Therefore, all passed file names are still there.

10. Regarding the program shown in the previous question...

a	It needs multiple turns for completing its execution, since each file being read requires a turn for its callback. File reads are asynchronous. Because of this, when they are completed, they execute a callback routine. That execution is run in the context of a separate thread, requiring an independent execution turn.
b	The "i" increase (i.e., i++) is incorrectly placed. It must be inside the callback body. No. If that statement is moved there, the loop will never end and an infinite number of reads will be started for the first passed file name.
c	This program shows and error and terminates if fewer than five file names have been passed as arguments. No. That behaviour only occurs when fewer than 3 file names have been passed. When 3 or 4 file names are given, the execution proceeds without showing any error message or aborting.
d	It prints the same length in all iterations. We need a closure for preventing this faulty behaviour from happening. No. The length is appropriately shown in every iteration, since it is computed using the "data" callback argument as a base. That argument carries the actual file contents. Therefore, the file length is correctly shown.

11. Regarding the mutual exclusion algorithms seen in Seminar 2, it is true that...

a	The central server algorithm correctly manages the situations in which that central server fails. False. When that central server fails, the algorithm cannot continue. Centralised algorithms have that critical problem.

b	The virtual unidirectional ring algorithm does not lose the token if the current process in the critical section fails. False. The current process in the critical section is the current holder of that token. If it fails, the token is lost.
c	The multicast algorithm with logical clocks uses fewer messages than the multicast algorithm based on quorums. No. Quorums were introduced in multicast-based algorithms for reducing their needed amount of messages.
d	The multicast algorithm with logical clocks complies with all 3 mutual exclusion correctness conditions. True. It satisfies safety, liveness and causal order.

12. Considering this program and knowing that it does not generate any 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=="e1"?++num1:++num2);
   console.log("Event "+arg+" has happened " + num + " times.");
   if (arg=="e1") setTimeout( function() {myEmit("e2")}, 3000 );
}

emitter.on("e1", listener);
emitter.on("e2", listener);
setTimeout( function() {myEmit("e1")}, 2000 );
```

The following sentences are true:

a	Event “e1” happens only once, 2 seconds after this program is started. Yes. The last line programs the generation of that event two seconds later. The listener for “e1” increases “num1” and shows “Event e1 has happened 1 times.” on the screen. Later, it programs the generation of event “e2” 3 seconds later. However, in the listener for “e2” there is no “setTimeout()” call. Because of this, “e1” is not generated again.
b	Event “e2” never happens. No. As we have explained in part “a”, event “e2” happens once.
c	Event “e2” happens periodically and its period is three seconds. No. As we have explained in part “a”, event “e2” happens once.
d	Event “e1” happens periodically and its period is two seconds. No. As we have explained in part “a”, event “e1” happens once.

13. Considering the program shown in the previous question...

a	The first event “e2” happens three seconds after the program is started. No. It happens 5 seconds after the program is started.

b	Since both events use the same listener, they both print messages with exactly the same contents when they happen. No. Their listeners are the same, but they receive one argument. That argument allows a different behaviour (and a different shown message) for each kind of event.
c	The first event "e2" happens two seconds after the first event "e1" has happened. No. That interval lasts three seconds.
d	None of its events happens twice or more times. True. Both events happen only once.

14. The ØMQ REQ-REP communication pattern is considered synchronous because...

a	Both sockets are connected or bound to the same URL. No. A PUSH-PULL communication channel also requires that both sockets share the same URL, but the resulting channel is asynchronous. Therefore, that feature is not a characteristic of synchronous channels.
b	Both sockets are bidirectional. No. ROUTER and DEALER sockets are also bidirectional and are able to define a communication channel, but they are asynchronous.
c	The REP socket uses a synchronous method for handling the received messages. No. All ZeroMQ sockets use a listener (i.e., a kind of callback) for handling message reception. Listeners have an asynchronous behaviour.
d	Once a message A is sent, both sockets cannot transmit their next sent message until they have received A's reply (REQ) or a new request (REP). True. This forces the agents that use REQ and REP sockets to use those sockets in a synchronous way, since this compels those sockets to deliver their messages following a sequential pipeline. Thus, a server cannot send a new reply until it has received its corresponding request. In the same way, a client cannot send a new request until it has received a reply for the current request.

15. Considering these two node.js programs...

```
// 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');
});
```

The following sentences are true:

a	The server terminates after sending its first reply to the first client. No. There is no execution termination statement in the 'data' event listener in the server program. That listener shows a message on the screen and replies to the client.

b	The client never terminates. False. When the client receives its first reply, it increases the value of "i". Then, it checks if that value is "1" (and it is). If so, it closes the connection. Once the connection is closed, there will be no pending event and the client process terminates. This doesn't cause the termination of the server, since the server is able to manage several connections.
c	This server may manage multiple connections. True. The createServer() method uses a connection handler as its callback. With that handler, it may manage multiple connections. To this end, that handler is called once per connection. Inside the callback, there are 'data' (for messages) and 'end' (for connection termination) event listeners.
d	This client cannot connect to this server. False. They can connect without problems when they are started in the same computer, since both refer to the same port (9000). The server is bound to that port using the "listen" operation, while the client is connected to the same port using the "connect" operation.

16. Leader election algorithms (from Seminar 2)...

a	...require the execution of a mutual exclusion algorithm, since their leader identity is placed in a shared resource and can only be updated by a critical section owner. False. Each process holds its leader identity in a private variable. Therefore, there is no need of any mutual exclusion algorithm.
b	...require consensus among all participating processes: all they must choose the same leader. True. That is the goal of these algorithms.
c	...do not need unique process identities. False. Unique process identities are needed for distinguishing each process from the rest.
d	...must respect causal order. False. Communication among processes does not demand causal order in the leader election algorithms presented in Seminar 2. That was a requirement in the solutions to the mutual exclusion problem in order to ensure fairness.

17. We want to implement a leader election algorithm using NodeJS and ØMQ, using the first algorithm explained in Seminar 2: the virtual ring algorithm. In order to implement this service, the best of the following options is...

a	Each process uses a REQ socket for sending messages to its successor in the ring and a REP socket for receiving messages from its predecessor. False. This algorithm uses unidirectional communication. This means that each process receives messages from its predecessor and forwards them (when needed) to its successor. REQ and REP sockets cannot be used for that kind of communication. They are bidirectional and will not allow sending a second message until they receive another message in the interim. Thus, REQ sockets will block in their second message sending attempt.

b	Each process uses a ROUTER socket for sending messages to its successor in the ring and a DEALER socket for receiving messages from its predecessor. No. Although both ROUTER and DEALER are asynchronous, a ROUTER socket is unable to send a message to another process A until it has received a message from A. At message reception, the ROUTER socket passes a first segment to that receiver process stating the identity of the other communication side. However, in this algorithm, communication is strictly unidirectional. This prevents ROUTER sockets from knowing the identity of their successor agent. Without such an identity, ROUTER sockets cannot send any message.
c	Each process uses a SUB socket for sending messages to its successor in the ring and a PUB socket for receiving messages from its predecessor. No. Although PUB and SUB sockets allow unidirectional communication, a SUB socket cannot be used for sending messages and a PUB socket cannot be used for receiving messages.
d	Each process uses a PUSH socket for sending messages to its successor in the ring and a PULL socket for receiving messages from its predecessor. True. This combination makes sense and it is the ideal one for that kind of unidirectional asynchronous communication.

18. We want to implement a leader election algorithm in NodeJS and ØMQ, using the second algorithm explained in Seminar 2: the bully algorithm. In order to implement the "election" (to ask better candidates about their liveness) and "reply" (to answer those "election" messages, confirming its liveness) messages of that algorithm, a feasible alternative for each participating process could be, assuming N processes:

a	A single connected REQ socket for sending the "election" messages to the other N-1 processes and receiving their "reply" answers and a single bound REP socket for managing the other communication side. False. "Election" and "reply" messages can be considered requests and replies, respectively. This suggests that they can be managed using REQ and REP sockets. In spite of this, the pattern described in this part is not correct. Each process has a unique REQ socket that is connected to the REP socket of every other process. In that scenario, messages sent through the REQ socket are handled in a round-robin way. In the algorithm, those messages must be sent ONLY to the processes with an identity higher than that of the sender. That behaviour cannot be achieved with a single REQ socket.

b	A single DEALER socket for sending “election” and “reply” messages to the other N-1 processes. That same socket is also used for receiving the incoming messages. False. DEALER sockets have the same problem described above for the REQ socket: they follow a round-robin policy for handling send actions.
c	N-1 PUSH sockets for sending both “election” and “reply”, when needed. A single SUB socket for receiving and handling those messages. False. PUSH sockets must interact with PULL sockets, instead of SUB sockets. There is no PUSH-SUB communication pattern.
d	A single bound PULL socket for receiving messages. N-1 PUSH sockets connected to the PULL of the other agents, for sending both “election” and “reply”, when needed. This scheme may work as intended. With N-1 PUSH sockets (one per each one of the other processes), the sender may choose which are the concrete targets of its “election” and “reply” messages. To this end, it must know which PUSH socket is associated to each process (for instance, at connection time, if each process is bound to a different and well-known port number). The PULL socket is used for receiving every incoming message. Both “election” and “reply” messages must include the identifier of their sender. For instance, in one of their segments: the second one. Therefore, each process has N-1 outgoing channels (from its N-1 PUSH sockets to the PULL sockets of the other processes) and N-1 incoming channels (from the PUSH sockets of the other N-1 processes to the local PULL socket). The incoming channel being used is determined by the segment of the received message that identifies its sender.

19. Which is the ØMQ socket type that uses multiple outgoing queues?

a	PUB sockets, for managing their broadcasts. No. There is a single outgoing queue in these sockets. Previously broadcast messages are lost by those subscribers that start late.

b	PUSH sockets, for managing multiple asynchronous unicast send() operations. False. There is a single outgoing queue in this type of sockets. Each message is only sent to a receiver. In case of multiple connections, outgoing messages are handled with a round-robin distribution policy.
c	REQ sockets, in case of being connected to multiple REP sockets. False. There is a single outgoing queue in this type of sockets. Each message is only sent to a receiver. In case of multiple connections, outgoing messages are handled with a round-robin distribution policy.
d	ROUTER sockets, using one outgoing queue per connection. True. Each outgoing queue is associated to a different connection identifier. Because of this, in send() operations, the ROUTER socket demands --as a first segment in the messages to be sent—the identity of the connection through which the message must be forwarded.

20. Considering these programs...

```
//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()]);
});
```

...and assuming that we have started one client and two servers using these commands:

$ node client & node server 8888 & node server 8889 &

The following sentences are true:

a	One server receives all requests with even values for “i” while the other receives all requests with odd values for “i”. True. When a REQ (or DEALER or PUSH) socket is connected to multiple URLs, it uses a round-robin (i.e., circular) policy to distribute its sent messages among all those connections. In this case, the REQ socket has been connected to two different servers. Since messages are numbered consecutively, one of the servers receives messages with odd numbers and the other receives messages with even numbers.

b	Every server receives, manages and replies all 100 requests. Thus, the client receives and prints 200 replies. No. Messages are not broadcast to all connected processes. Each message is sent through a single connection to a single server. This means that each server receives 50 messages and that the client receives 100 replies.
c	The first few requests are lost since the client has been started before the first server was started. No. A REQ socket does not transmit a new request until the reply to the current request has been received. Besides, the first request is kept in the outgoing buffer until the intended first connection is correctly set.
d	If one of the servers fails in the middle of the execution, the client and the other server are able to manage without blocking all their remaining requests and replies. No. The client remains indefinitely blocked waiting for a reply that will never arrive (i.e., the first reply that the crashed server has been unable to send). This prevents the client from transmitting any other requests. As a result, the other server remains also blocked waiting for more requests, but those requests will never arrive. Therefore, both processes remain alive, but they are unable to advance.