A

This exam consists of 20 multiple choice questions. In every case only one of the answers is correct. You should answer in a separate sheet. If correctly answered, they contribute 0,5 points to the final grade. If incorrectly answered, the contribution is negative: -0.167. So, think carefully your answers.

THEORY

1. The PaaS service model in cloud computing…

a	…is centred in hardware virtualisation. False. Hardware virtualisation is a problem that should be managed at the infrastructure layer. The *Platform as a Service (PaaS)* model is placed on top of that infrastructure management (IaaS) layer.
b	…provides distributed applications to its users. False. The *Software as a Service (SaaS)* model is responsible of that kind of service. PaaS is placed below SaaS in that logical layered architecture.
c	…manages the underlying infrastructure and automates the deployment and administration tasks to application providers. True. Those are the tasks to be considered in a PaaS model.
d	…avoids concurrency as much as possible. False. Cloud service models need not avoid concurrency. Note that all distributed systems and services are examples of concurrent systems. Concurrency is inherent in distributed services. All cloud service models are also distributed service models. Therefore, they are concurrent.

2. Wikipedia…

a	…is an example of distributed application following the peer-to-peer interaction approach. Therefore, it is highly scalable. False. Wikipedia uses a client/server interaction approach.
b	…uses web browsers as a particular type of server agent. False. Web browsers are examples of client agents.
c	…stores its persistent information in the client nodes. False. Its persistent information is maintained in databases at the server side, not in the client nodes.
d	…uses reverse proxies in order to increase its throughput. True. This is one of the techniques explained in Unit 1 for improving throughput and scalability. Reverse proxies have been used in Wikipedia since its first releases.

3. LAMP stands for:

a	Light and Available Multi-Processing. False.

b	Linux, Apache, MySQL and PHP. True. LAMP systems were explained in Unit 1 (in its Wikipedia section) in order to describe the regular modern architectures for web servers.
c	Linux, Acrobat, MongoDB and Python. False.
d	Linux, Apache, Memcache and PostgreSQL. False.

4. A system model...

a	...sets the architecture of the distributed system. False. The architecture of a distributed service is not defined in its system model.
b	...takes care of and describes thoroughly the communication hardware and the communication protocols to be used in the real system. False. The communication hardware is not described in a system model.
c	...provides a high-level view of a system, taking care of the main properties to be considered when we design algorithms and protocols for that system. True. This is a valid summary of the definition of system model given in Unit 2.
d	...specifies all the details and algorithms being used in a middleware. False. No details about any specific software element are specified in a system model. Only the main abstract properties are considered (those that are relevant for writing algorithms).

5. Regarding time and synchronisation...

a	A distributed system model should be as synchronous as possible since synchrony avoids race conditions. False. Not all kinds of synchrony are related to race conditions. Besides this, the more synchrony we assume in a system model the more difficult will be to provide the behaviour conditions being assumed in that system model.
b	Synchronisation should be minimised since it hampers throughput and scalability. True. Synchronisation mechanisms will block agents activity at some times. Because of this, they are not convenient for achieving good levels of throughput and scalability.
c	Synchronous communication never blocks processes. False. Synchronous communication blocks sender processes until they get some kind of acknowledgement from the receiver side.
d	Local clocks in every node may be synchronised without exchanging messages. This is achieved using Cristian's algorithm. False. Node clock synchronisation (with both kinds of clocks: logical and "physical") always demands some kind of external interaction; i.e., some messages are needed to this end. Indeed, the algorithm from Flaviu Cristian (one of the eldest and most well-known) needs message exchange with a clock server.

6. In the asynchronous programming paradigm...

a	There are multiple threads per process. False. Multiple threads per process are not mandatory (nor a regular characteristic) in asynchronous programming.

b	All events happen simultaneously. False. Different events may happen at different times.
c	If multiple events happen at a time, then their actions are enabled but they will be executed sequentially and atomically. True. All actions associated to incoming events are atomic (i.e., they are executed in a single turn without being interrupted) and, due to this, they will be executed one after the other.
d	Programmes cannot be based on callbacks. False. They can be based on callbacks and we have used in this subject a language (JavaScript) that uses this approach.

7. Every message-oriented communication middleware...

a	...is an example of authentication service (i.e., a security-related service). False. Authentication services do not provide a message-brokering interface in the regular case; i.e., they are not any example of communication middleware.
b	...is not compelled to provide location transparency. True. Message-oriented communication middleware is aimed at improving communication throughput. Location transparency is not a goal in this case. We have seen an example in TSR: ØMQ. In that middleware the bind() and connect() calls are not location-transparent since the programmer must state the IP address and port. In order to be location-transparent the communication software should use names (instead of addresses) to refer to agents.
c	...uses proxies and skeletons in order to enhance its throughput. False. Proxies and skeletons are used in remote method invocation approaches, providing agent/object interfaces. Message-oriented middleware does not use those kind of interfaces. It regularly uses send() operations provided by sockets.
d	...uses centralised algorithms and depends on a central broker. False. Message-oriented communication middleware may be either brokered or brokerless; i.e., it does not depend always on a central broker. Even in the case of using a broker, that broker does not need many steps (i.e., a non-trivial algorithm) for managing agent-to-agent communication. Moreover, scalable message-oriented communication is brokerless in most cases, as the ØMQ library has shown.

8. Transient (i.e., non-persistent) messaging…

a	…blocks the sender until the receiver reports that it has delivered the message. False. That is not the definition of transient messaging but of synchronous communication.

b	…provides location transparency. False. The degree of persistency in a communication approach is not related to location transparency. In order to provide location transparency we need communication stubs at both domains (sender and receiver) in order to provide local interfaces that match the interface of the other remote agent.
c	…ensures failure transparency. False. The degree of persistency in a communication approach is not related to failure transparency. In order to provide failure transparency we should replicate both sender and receiver agents (when we want to hide process failures) and retry sending actions (when we want to overcome message losses).
d	…prevents communication from starting until both sender and receiver are ready for managing message transmission. True. Transient communication uses channels that are unable to hold the messages being propagated. Because of this, both communication agents should be ready in order to start any message transmission.

SEMINARS

9. Considering this programme:

```
var fs=require('fs');
if (process.argv.length<5) {
      console.error('More file names are needed!!');
      process.exit();
}
files = process.argv.slice(2);
files.forEach( function (name) {
      fs.readFile(name, 'utf-8', function(err,data) {
            if (err) {
                  console.log(err);
            } else
            console.log('File '+name+': '+data.length+' bytes.');
      })
})
console.log('We have processed '+files.length+' files.');
```

The following sentences are true when no error arises in the programme execution:

a	This JavaScript programme needs at least 5 file names as arguments from the command line. False. It needs at least 5 arguments but the first argument is always “node” (the name of the interpreter) and the second is always the name of the programme to be run. Therefore it only requires 3 file names as a minimum (instead of 5, as it is said in this sentence).

b	It usually prints the name and length for each one of the file names received from the command line. True. This is the functionality it implements.
c	It prints "We have processed N files" at the end of its execution, being N the number of file names given as arguments. False. The statement that prints that message is at the end of the programme but JavaScript is an asynchronous programming language. As a result, that statement is run in the first turn, while all other console.log() statements are placed in the readFile() callback and will be executed in subsequent turns. This means that the message being mentioned in this part of the question is printed at the beginning of the programme execution.
d	This programme can be executed in a single turn. False. It needs multiple turns, as it has been described in the explanation of part "c".

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

a	It is unable to print the correct file name in each iteration. Instead of this, it always prints the name of the last file. False. The file name is correctly managed in the callback of the readFile() function since the readFile() call itself is contained in the callback for the forEach() loop and the file name is being received as an argument in that encompassing function. Thus, this schema is similar to a closure and it guarantees that each invocation of the readFile() callback is made inside a scope that contains a valid value for the "name" variable.
b	It generates an exception and crashes if any error happens when it tries to read a file. False. If any error happens at that point, it will be communicated through the "error" parameter of the readFile() callback. In that case, simply prints an error message and no exception is rised.
c	In case of error, it prints a message to the screen reporting that error and goes on. True. We have already explained this scenario in the justification of part "b".
d	It always prints the same length in all iterations. We need a closure in order to avoid this erroneous behaviour. False. As we have explained in part "a", that closure is already in use in the current programme.

11. Some safety conditions for all mutual exclusion algorithms seen in Seminar 2 are...

a	A process enters the critical section when it completes the CS-exit protocol. False. When a process completes its CS-exit protocol it has exited the critical section. Therefore, it is in its remaining section at that point (i.e., outside the critical section).

b	Processes must use asynchronous communication channels. False. None of the algorithms being presented in that seminar required asynchronous communication channels.
ℂ	There cannot be more than one process executing the critical section. True. This is the correctness requirement for valid solutions to this problem. Note that safety conditions should state which are the constraints to be respected by correct solutions. In other words, safety conditions guarantee that nothing incorrect or invalid happens while the tasks described in the algorithm are running.
d	No starvation: there is a limit on the amount of times that a process in the enter-CS protocol yields pass to other processes. False. This is a liveness condition, not a safety condition.

12. Considering this programme...

```
var ev = require('events');
var emitter = new ev.EventEmitter;
var num1 = 0;
var num2 = 0;
function emit_e1() { emitter.emit("e1") }
function emit_e2() { emitter.emit("e2") }
emitter.on("e1", function() {
   console.log( "Event e1 has happened " + ++num1 + " times.")});
emitter.on("e2", function() {
   console.log( "Event e2 has happened " + ++num2 + " times.")});
emitter.on("e1", function() {
   setTimeout( emit_e2, 3000 )});
emitter.on("e2", function() {
   setTimeout( emit_e1, 2000 )});
setTimeout( emit_e1, 2000 );
```

The following sentences are true:

a	Event “e1” happens only once, 2 seconds after this programme is started. False. Event “e1” happens, in its first time, 2 seconds after the programme is started. Afterwards, it happens every 5 seconds. Therefore, it doesn’t happen only once.
b	Event “e2” never happens. False. Event “e2” happens 3 seconds after each occurrence of event “e1”. Therefore it will happen (approximately) as many times as event “e1”.
ℂ	The period of “e2” is five seconds. True. Note that “e1” happens 2 seconds after each occurrence of event “e2” and “e2” happens 3 seconds after each occurrence of event “e1”. Due to this, both events are happening every 5 seconds.
d	The period of “e1” is three seconds. False. Its period is five seconds.

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

a	No message is printed in its execution. False. Each time events “e1” and “e2” occur, a message is printed showing how many times each event has happened up to that point.

b	No event is generated in its execution, since its emit() calls are incorrect. False. All the emit() calls are correctly used and both events are happening periodically without any trouble.
c	We cannot have more than one listener for each event. Therefore, the programme is immediately aborted by an exception. False. There may be multiple listener functions per event. No exception is rised due to this fact.
d	The programme correctly reports each event occurrence. True. The explanations have been given in the remaining parts of this question and in the previous question, too.

14. In a ØMQ REQ-REP communication pattern, the REP socket is bound and the REQ socket is connected to it. However, other variants might also work...

a	Both REQ and REP sockets can be bound to the same address and port. False. In that case an "address already in use" error will be generated and the second bind() attempt will fail.
b	The REQ socket is bound and the REP socket is connected to it. True. This is an alternative that may also work as we saw in different activities from Seminar 3.
c	No socket needs to be bound. Both connect to the same address and port. False. One of the sockets must use the bind() call. In order to define a communication channel one of the agents should be bound to a given URL and the other (or others) should connect to it.
d	No change is admitted. The REP socket needs to be always bound and the REQ socket needs to connect later to it. False. Look at part "b" for an explanation.

15. Considering these two node.js programmes...

```
// 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 client = net.connect({port:
  9000}, function() {
    client.write('Hello ');
  });
client.on('data', function(data) {
  console.log('Reply: '+data);
  client.end();
});
client.on('end', function() {
  console.log('client ' +
    'disconnected');
});
```

The following sentences are true:

a	The client terminates after sending a request and receiving its reply. True. To this end, it is closing its connection to the server using the end() call once it has received the answer to its single request.

b	The server terminates after sending its first reply to the first client. False. The function that the programmer has passed as a callback in the createServer() call is able to manage a connection. That callback is used as many times as client connections have been settled. Moreover, the server will not terminate. It will remain alive waiting for new connections.
c	This server can only handle a connection. False. Connection handling has been already explained in the justification of the previous part.
d	This client cannot connect to this server. False. It can connect without problems if the server has been already started when the client is run.

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

a	...have no safety condition. False. All distributed algorithms have at least one safety condition. Indeed, we presented four complementary safety conditions for these algorithms (e.g., (1) the leader must be unique, (2) all processes elect the same leader, (3) once elected, the leader doesn't change while no failure arises, (4) once elected, all participants terminate the algorithm).
b	...are usually started when the current coordinator being used in another algorithm fails. True. That is the regular event that starts these algorithms.
c	...assume, in their system models, that failures never happen. False. If no failure arises, there is no need to elect any new leader. Note that this would contradict the third safety condition mentioned in our explanation in part "a".
d	...need a consistent snapshot of the current global system state (taken, e.g., by the Chandy & Lamport algorithm) in order to be started. False. Each participating process manages its own state. There is no need to use any global state in order to start.

17. We want to implement a chat service using node.js and ØMQ. The server multicasts the user messages and should never block trying to send a message (of any kind). The client programmes send user messages to the server, wait for user messages forwarded by the server and report to the server when a user enters or abandons the system. In order to implement this chat service...

a	The server needs a DEALER and a ROUTER socket to balance the load among clients. False. No socket of those kinds allows message multicasting to the clients. Instead of these, the server needs a PUB socket for multicasting its forwarded messages and a PULL socket for receiving the client messages to be forwarded.

A

b	Each client needs a PULL socket to interact with the server. False. The receiving socket to be used by clients should be a SUB socket due to what has been explained in part "a".
c	Each client needs a REP socket to interact with the server. False. The receiving socket to be used by clients should be a SUB socket due to what has been explained in part "a".
d	Each client needs a SUB socket to interact with the server. True. The receiving socket to be used by clients should be a SUB socket due to what has been explained in part "a".

18. Considering these programmes...

```
//client.js
var zmq=require('zmq');
var so=zmq.socket('dealer');
so.connect('tcp://127.0.0.1:8888');
so.send('request');
so.on('message',function(req,rep){
  console.log("%s %s",req,rep);
});
```

```
// server.js
var zmq = require('zmq');
var rp = zmq.socket('rep');
rp.bindSync('tcp://127.0.0.1:8888');
rp.on('message', function(msg) {
  console.log('Request: ' + msg);
  rp.send([msg,'answer']);
});
```

The following sentences are true:

a	The server is unable to receive and process any message from this client. True. The server uses a REP socket. A REP socket expects that the incoming messages have an empty delimiter in any of their initial segments. The client uses a DEALER socket to interact with the server and the messages being sent to that server do not have any explicit delimiter. Since DEALER sockets do not prepend any empty delimiter (while REQ sockets do this automatically and in a transparent way for the programmer), this implies that no message could be received by this server.
b	The server may obtain the messages sent by this client if the former is started before starting the latter. False. With the currently presented programmes, no communication is possible. This has already been explained in the justification of part "a".
c	The client prints "request answer" to the screen once it has sent its 'request' message. No problem arises in order to do this. False. With the currently presented programmes, no communication is possible. This has already been explained in the justification of part "a".
d	The server cannot work. Instead of bindSync() we should use bind() in order to fix this problem. False. With the currently presented programmes, no communication is possible. The reason of that lack of communication has already been explained in the justification of part "a". It is not related to the bind() or bindSync() operations.

19. Considering these programmes...

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

A

The following sentences are true:

a	Both client and server exchange messages in a synchronous way in this example, since they follow a request-reply pattern. False. Despite sending multiple requests from the client process and returning multiple replies from the server, the communication isn't synchronous since both processes are using DEALER sockets. DEALER sockets are asynchronous. Because of this, the client could have sent all its requests before receiving its first reply, for instance, in case of finding a server that was overloaded by other clients.
b	The server returns a message with 2 segments to the client. The second segment contains a value that is 3 times greater than that in the first segment. True. That is the behaviour being implemented in the "message" listener in the server programme.
c	The client sends 100 requests to the server. False. It only sends 99 requests. The numbers being used start with 1 and end with 99.
d	The client prints to the screen the following sequence: "Sending 1", "1 3", "Sending 2", "2 6", "Sending 3", "3 9"... False. As we have explained in the justification of part "a" nothing guarantees that each reply is interleaved between two consecutive request messages. Therefore the "Sending..." messages and the printed replies won't be interleaved as they have been shown in the sequence presented in this part.

20. Please consider which of the following variations will generate new programmes with the same behaviour as that shown in question 19...

a	The 'rq' socket should be of type 'REQ' and the 'rp' of type 'REP'. False. In that case we would obtain a synchronous behaviour instead of the asynchronous one shown in question 19.
b	The 'rq' socket should be a 'PUSH' and 'rp' should be a 'PULL'. False. In that case only a unidirectional communication (from client to server) would be possible, without admitting any server reply.
c	Both processes need two sockets, building a PUSH->PULL channel from client to server (requests) and a PUSH->PULL channel from server to client (replies). True. In this way we'll implement two unidirectional and asynchronous channels, matching appropriately the behaviour of a DEALER-DEALER channel.
d	The 'rq' socket should be a 'PUB' and 'rp' should be a 'ROUTER'. False. PUB sockets are unidirectional. They don't admit any message reception, preventing the client from receiving any server reply.