NIST **First Partial** **Units 1,2,3 (1st half) Lab 1** **2022-11-03** **A**

This exam is worth 10 points, and consists of 24 questions. Each question poses 4 alternatives and has only one correct answer. Each correct answer earns 10/24 points, and each error deducts 10/72 points. You must answer on the answer sheet.

1 *In cooperative computing:*

- **a** The computing tasks are usually carried out in the *clients*, while the *server* distributes the tasks to be carried out.
- **b** A client failure will prevent global computation from terminating.
- **c** Clients often exchange partial results with each other.
- **d** All other options are true.

2 *When analyzing the lifecycle of services, the* system administrator *role focuses on:*

- **a** Developing applications requested by users.
- **b** Deciding which components will be part of a distributed application that provides the requested service.
- **c** Deciding on which nodes each component will be installed and with what configuration, as well as verifying that each computer works correctly.
- **d** None of the other options is true.

3 *An advantage of the asynchronous programming model when it is compared to concurrent programming based on multiple threads is:*

- **a** Greatly reduced chance of race conditions.
- **b** Improves performance: not only does it launch different threads, but these threads are never in the WAITING state.
- **c** By using non-persistent communication, clients do not need to connect to any server before sending their requests.
- **d** All other options are valid.

4 *In Wikipedia, component replication …:*

- **a** Is not necessary.
- **b** Is commonly used in its components, combined with other mechanisms such as the use of caches and the distribution or sharding of data.
- **c** Is used only in the *Apache* component.
- **d** Is used only in the *MySQL* component.

5 *Let us consider the following JavaScript program:*

```
const fs=require("fs")

console.log("Call to asynchronous readFile")
fs.readFile("/proc/loadavg",(e,d)=> {
  if (e) console.error(e.message)
  else console.log(d+"")
})
console.log("End of asynchronous readFile")

console.log("Call to synchronous readFile")
console.log( fs.readFileSync("/proc/loadavg") + "" )
console.log("End of synchronous readFile")
```

What is the last string that this program shows, if the file to be read exists, has content, and there are no errors in its reading?

- **a** `End of asynchronous readFile`
- **b** The content of the file `/proc/loadavg`
- **c** `End of synchronous readFile`
- **d** None of the other options is correct.

Consider the following set of JavaScript programs:

```
// Program: ex1.js
const ev = require('events')
const emitter = new ev.EventEmitter()
const e1 = "print"
emitter.on(e1, function(num) {
  return () =>
    console.log("Event " + e1+ ": " + ++num)
}(0))
emitter.emit(e1)
```

```
// Program: ex2.js
function f(x) {
 return function (y) {
   let z = x + y
   return 20
 }
}
```

```
// Program: server.js
const net = require('net')
let server = net.createServer(
 function(c) {
  console.log('server connected')
  c.on('end', () =>
   console.log('server disconnected') )
  c.on('error', () =>
   console.log('some connect.error') )
  c.on('data', function(data) {
   console.log('data from client: ' + data)
   c.write(data)
  })
 }) // End of net.createServer()
server.listen(9000, () =>
console.log('server bound') )
```

6 *What does* `ex1.js` *show on the screen?*

a It shows Event print: 1

b It shows a succession of messages:
event print: 1
event print: 2
event print: 3
…

c Line `emitter.emit(e1)` throws an exception.

d No message is displayed.

7 *How many anonymous functions (not to be confused with function calls) are defined in* `ex1.js`*?*

a None of the other options is true.

b Infinite, since a recursive definition is given.

c One.

d Two.

8 *What value is returned by a call to function* `f` *with argument* `10`*, i.e.* `f(10)`*, in program* `ex2.js`*?*

a undefined

b 10

c A function.

d NaN

9 *Can the* `server.js` *program support multiple connections from client processes?*

a It can't, because JavaScript only has one thread of execution and each connection to a client needs to be handled by a different thread.

b It can, because both the establishment and closing of the connection, as well as the arrival of new messages or the occurrence of errors in the connection are modeled as events and JavaScript has a queue of events to manage them.

c It can't, because there is only one callback to manage the establishment of a connection and while it is running, there can be no other running activity.

d In principle it could, but this program can't because it doesn't have any `listener` for the `connect` event.

10 *Consider that there is a function* `countLines(text)` *that returns the number of lines contained in the given text string in its* `text` *parameter. If you needed to write a Node.js program that uses promises and displays the number of lines contained in the* `Example.txt` *file in the current directory (existing and with read permission), a possible solution would be:*

a None, because with promises it is not possible to carry out this management.

b

```
const fs=require('fs')
function countLines(t) { return t.split('\n').length }
console.log(countLines(
  fs.readFileSync("Example.txt",'utf-8'))
```

c

```
const fs=require('fs').promises
function countLines(t) { return t.split('\n').length }
fs.readFile("Example.txt",'utf-8')
.then(countLines).then(console.log)
```

d

```
const fs=require('fs').promises
function countLines(t) { return t.split('\n').length }
fs.readFile("Example.txt",'utf-8').then(console.log)
```

11 *Choose the correct statement in regard to Node.js:*

a Node is implemented internally using a single thread of execution. That is, Node itself is single-threaded.

b Node integrates the V8 JavaScript engine, the same JavaScript engine used by Google Chrome.

c Node is suitable for running applications made in JavaScript and Java.

d Node and all the functionality of its modules is integrated into modern web browsers, such as Google Chrome or Firefox.

12 *If we run the following code snippet, choose the correct statement:*

```
const ev = require('events');
const emitter = new ev.EventEmitter
emitter.on("event1",(x) => console.log(x));
emitter.emit("event1", "hello");
emitter.emit("hello");
console.log("bye")
```

a The string `hello` will be printed first on the console and then `bye` will be printed

b It will cause an error, because the `hello` event is emitted and there is no listener for that event.

c The string `bye` will be printed first on the console and then `hello` will be printed.

d If we execute the program several times, sometimes `hello` will be printed before `bye` and sometimes `bye` will be printed before `hello` .

13 *Regarding the broker (or manager) in messaging systems, select the true statement:*

a Brokerless systems are not suitable when weak persistence is desired

b A brokered system can be built from a non-brokered one

c Brokered systems facilitate, but do not directly provide, a shared state view

d A system without a broker can be built from another with a broker

14 *ØMQ, as a messaging system, tries to cover several goals. Choose which one is NOT a goal:*

a Ease of use for programmers by reproducing an API similar to BSD sockets

b Reliability guaranteed by automatic tracking of messages with a pending receipt

c Reuse of the same code, except the URL, to communicate threads, processes on the same computer, and processes on different computers

d Ease of solving common problems by supporting basic communication patterns

15 *This is a sender that connects without errors to receivers that have performed the corresponding bind operation with a ØMQ REP socket:*

```
const zmq = require('zeromq')
const q = zmq.socket('req')
q.connect('tcp://10.0.0.1:8887')
q.connect('tcp://10.0.0.2:8888')
q.connect('tcp://10.0.0.3:8889')
```

If the sender then attempts to execute these 3 statements:

```
q.send('Hello A')
q.send('Hello B')
q.send('Hello C')
```

This will happen:

- **a** Each receiver receives the 3 messages in the same order in which they have been sent
- **b** The sender cannot initiate the invocation of the second q.send until receiving a response for the first one
- **c** Each receiver receives only one of the messages, but the sending socket must wait for a response for each one before propagating the next one
- **d** Each receiver receives only one of the messages, and the sending socket should only wait for a response after propagating the last of the three

16 *Under what conditions several ØMQ sockets of type REQ can be interconnected?*

- **a** Only when there is exactly one performing the bind operation
- **b** Only when there is no more than one simultaneous sending in those sockets
- **c** Only when using a local transport (IPC or threads)
- **d** They cannot be interconnected because REQ requires that a complementary socket be used at the other agent

17 *A server program has been developed that uses a REP socket to interact with clients, which use REQ sockets. However, this solution only allows one request to be processed at a time and its performance should be improved. To do this, a programmer proposes that both clients and server use a single PUSH socket to send messages and a single PULL socket to receive them. If there will usually be multiple clients connected and interacting with the server, is this a correct solution?*

- **a** No, because the PUSH socket cannot be used to send messages: it only allows their reception.
- **b** No, because the PUSH socket sends messages following the connection rotary order and could send responses to clients that have not sent any request.
- **c** Yes, because the PUSH/PULL pattern does not block the sending or receiving of messages. Thus, the performance will clearly improve.
- **d** None of the other statements is true.

18 *On 0MQ, on establishing a connection:*

- **a** Each socket can only do, at most, one bind operation.
- **b** Each socket can only do, at most, one connect operation.
- **c** The bind operation must always precede the connect operations.
- **d** The other answers are wrong.

19 *Sources of complexity in a distributed system include:*

- **a** The request for services.
- **b** The information format and structure.
- **c** Fault management.
- **d** The other answers are valid.

20 *Indicate the result that will be seen on the console when executing the following code fragment:*

```
function g(a) {
  a = a + 1
  return (b) => { return b + a }
}
let f = g(1)
console.log ( f(1) + g (1)(1) )
```

a undefined

b 6

c 7

d NaN

21 *Take the following code snippet and indicate, in the first 4 seconds (inclusive), how many times the text* `Event one treated` *will be displayed on the screen.*

```
const ev = require('events')
const emitter = new ev.EventEmitter()
const t = 1000, e1 = "one"

var handler = function() {
   console.log("Event "+e1+" handled")
}

emitter.on(e1, handler)

function stage() {
  emitter.emit(e1)
  setInterval(stage, t)
}

stage()
```

a None because there is an error in the function provided as an argument to emitter.on

b None because there is an error in the function provided as an argument to setInterval

c 4 or 5

d At least 7

22 *Given the following program, choose the correct statement:*

```
let MAX=1000

console.log("one")
setTimeout ( () => console.log("two"), 100)

// Do work
for (let i=0; i<MAX; i++) {}

setTimeout ( () => console.log("three"), 99)
setTimeout ( () => console.log("four"), 0)
console.log("five")
```

a When we run it, we will always see `five` before `four` .

b When we run it, we might see `four` before `five` .

c When we run it, we will always see `two` before `three`

d The program prints `one` and `five` and ends without printing anything else.

23 *Let us consider the following snippet of the reverse proxy code seen in the third session of Lab 1:*

```
const net = require('net')
const ...

const server = net.createServer(function (s2) {
 const s1 = new net.Socket()
 s1.connect(parseInt(REMOTE_PORT),
REMOTE_IP, function () {
     ***
  })
}).listen(LOCAL_PORT, LOCAL_IP)
console.log("TCP server accepting connection
on"+
" port: " + LOCAL_PORT)
```

*The missing statements on the *** lines are:*

a s2.on('data', function (m2) {s1.write(m2)})
s1.on('data', function (m1) {s2.write(m1)})

b s1.on('data', function (m1) {s2.write(m1)})
s2.on('data', function (m2) {s1.write(m2)})

c None of the other options are valid.

d The two options that show code are valid.

24 *The programmable proxy exercise includes a new proxy service that, in short, allows you to modify REMOTE_PORT and REMOTE_IP. Imagine that for this it is proposed to incorporate the following code to the programmable proxy:*

```
// s1 connects the proxy to the client.
// s2 connects the proxy to the remote server.
const prog = net.createServer((s3)=>{
  s3.on('data', function(m3) {
    ***
  })
})
```

And that the request sent by programmer.js is constructed from argv as follows:

```
s.write(JSON.stringify({
  "remote_ip": argv[2],
  "remote_port": parseInt(argv[3])
}))
```

*The missing instructions on the *** lines of the proposed programmable proxy are:*

a

```
s2.end();
REMOTE_IP = JSON.parse(m3).remote_ip;
REMOTE_PORT = JSON.parse(m3).remote_port
```

b

```
s1.end();
REMOTE_IP = JSON.parse(m3).remote_ip;
REMOTE_PORT = JSON.parse(m3).remote_port
```

c

```
REMOTE_IP = JSON.parse(m3).remote_ip;
REMOTE_PORT = JSON.parse(m3).remote_port
```

d

```
REMOTE_IP = m3.remote_ip;
REMOTE_PORT = m3.remote_port
```

NIST **First Partial** **3 November 2022** **A**

Fill and deliver this answer sheet. Each question has a single correct answer. Don't forget to correctly write your personal data

Don't fix a wrong answer with another mark: erase it or cover it with Typex

A question with more than an answer will be discarded

0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	4
5	5	5	5	5	5	5	5
6	6	6	6	6	6	6	6
7	7	7	7	7	7	7	7
8	8	8	8	8	8	8	8
9	9	9	9	9	9	9	9

DNI: ______________________________

Surname: ______________________________

First Name: ______________________________

1	A	B	C	D	**21**	A	B	C	D
2	A	B	C	D	**22**	A	B	C	D
3	A	B	C	D	**23**	A	B	C	D
4	A	B	C	D	**24**	A	B	C	D
5	A	B	C	D					
6	A	B	C	D					
7	A	B	C	D					
8	A	B	C	D					
9	A	B	C	D					
10	A	B	C	D					
11	A	B	C	D					
12	A	B	C	D					
13	A	B	C	D					
14	A	B	C	D					
15	A	B	C	D					
16	A	B	C	D					
17	A	B	C	D					
18	A	B	C	D					
19	A	B	C	D					
20	A	B	C	D					