This exam consists of 20 multiple-choice questions. In every case only one of the answers is the correct one. Answers should be given in a SEPARATE answer sheet you should have been provided with.

All questions have the same value. If correctly answered, they contribute 0,5 points to the final grade. If incorrectly answered, the contribution is negative, equivalent to 1/5th the correct value, which is -0,1 points. So, think carefully your answers. In case of doubt, leave them blank

The exam can be completed within 1 hour.

1. The command "`git push origin master`"

	Brings all commits from the remote identified as "`origin`" to our local repository
	Deletes the current working directory
	Always copies the local commits into the remote repository
	When successful, sets the remote master branch pointing to the same commit as the local master branch
	All the above.
	None of the above.

2. Both commands "`git pull`" and "`git fetch`"

	Are equivalent
	Bring all commits of a remote repository to the local repository
	Set the local master branch pointing to the same commit as the remote master branch.
	Modify the working directory.
	All the above.
	None of the above.

3. Gitlab is...

	A concrete server within the UPV
	The name of a repository
	A way of committing changes to a repository
	An experimental version of Git
	All the above.
	None of the above.

4. In the development workflow you are asked to use with GitLab …

	Every developer in the team must have a private remote repository
	There is a canonical repository that every member of team can read, different from any of the private remotes.
	Only the owner can write to her private remote repository
	Only one member of the team should write to the canonical repository
	All the above.
	None of the above.

5. Within a TSR team, each private remote GitLab repository…

	Should be accessible only by its owner
	Should be writeable by its owner
	Should be writeable by the integration manager
	Should be writeable by the teacher
	All the above.
	None of the above.

6. A local Git repository…

	Must be associated to a remote Git repository
	Can be associated to only one remote repository
	Can be cloned onto another local repository.
	Cannot be modified
	All the above.
	None of the above.

7. Select the command that creates an empty local Git repository…

	"`git init; touch README; git commit`"
	"`git clone git:init`"
	"`git init origin master`"
	"`git init -u origin master`"
	All the above.
	None of the above.

8. Assume the following situation: Repository R (whose URL is R_{url}) can be accessed by two developers D1, and D2. R's master branch points to commit C, containing file "`foo.js`", with a single line, "`var fs = require( 'fs' )`". Each developer executes "`git clone` R_{url}`; cd R;`" at their work machine. D1 edits "`foo.js`" and adds line "`fs.readFileSync( 'myfile' )`". D2 also edits "`foo.js`" but adds line "`fs.readFileSync( 'nofile' )`". Both commit their changes, at their own pace, obtaining commits C1 and C2, respectively. Then they each execute "`git push`".
Then we can assure that…

	Nothing changes at R
	The master branch at R will point to C1
	The master branch at R will point to C2
	One or more developers get an error when executing "`git push`"
	All the above.
	None of the above.

Consider code snippet 1:

```
var net = require('net');
var server = net.createServer(  function(c) {
   console.log('server connected');
   c.write('World');
   c.end();
});

server.listen(8000, function() {
  console.log('server bound');
});
```

9. **Assume process P1 runs the code in snippet 1. Let P2 be another process started after P1 on its same host. Further assume P2 connects to port 8000. Then …**

	P1 sends an answer message to P2 after receiving the request message.
	When P1 starts to listen it outputs the message '`server connected`'.
	P2 may receive the message "World" before sending any information to P1.
	P1 does not necessarily disconnect when P2 closes its socket.
	All the above.
	None of the above.

Consider code snippet 2:

```
var net = require('net');

var server_port = process.argv[2];
var text        = process.argv[3].toString();
var client      = net.connect({port: parseInt(server_port)}, function() {
   console.log('client connected');
   // This will be echoed by the server.
   client.write(text);
});
client.on('data', function(data) {
  console.log(data.toString());
  client.end();
});

```

10. **Assume process P1 runs the code in snippet 2. Let P2 be a server process listening for connections on port 8000. Then …**

	P1 outputs a console message when P2 closes its socket.
	P1 can connect to P2 even though P1 and P2 are on different hosts.
	If P2 is not running, the message in variable '`text`' waits within the output buffer until P2 runs.
	We could invoke P1 from the command line like this `$ node client 127.0.0.1 8000 hello`
	All the above.
	None of the above.

Consider code snippet 3:

```
var net = require('net');

var LOCAL_PORT  = 8000;
var LOCAL_IP  = '127.0.0.1';
var REMOTE_PORT = 80;
var REMOTE_IP = '158.42.156.2';

var server = net.createServer(function (socket) {
    socket.on('data', function (msg) {
       var serviceSocket = new net.Socket();
       serviceSocket.connect(parseInt(REMOTE_PORT), REMOTE_IP, function (){
           serviceSocket.write(msg);
       });
       serviceSocket.on('data', function (data) {
           socket.write(data);
       });
       console.log("Client connected");
    });
}).listen(LOCAL_PORT, LOCAL_IP);
console.log("TCP server accepting connection on port: " + LOCAL_PORT);
```

Assume this code is run to obtain process P1. Data from clients connecting to port 8000 is supposed to arrive in chunks through the connection they establish. Each chunk raises a 'data' event in the socket representing the connection.

Let process P2 be launched after P1 on P1's computer, such that P2 connects to the local port 8000. Answer the following 3 questions.

11. When P2 connects to local port 8000, P1 …

	… connects to REMOTE_IP.
	… prints '`Client connected`' to the console.
	… starts listening for data from REMOTE_IP.
	… stops accepting new connections.
	All the above.
	None of the above.

12. Let P2 send two large chunks of data in sequence, D1 and then D2, through its connection to P1.
Then, in the absence of failures, …

	D2 always arrives to REMOTE_IP after D1
	D1 always arrives to REMOTE_IP after D2
	Only D1 is relayed to REMOTE_IP
	Only D2 is relayed to REMOTE_IP
	All the above.
	None of the above.

13. Assume the server at REMOTE_IP can only handle one connection at a time.
Then, if P2 sends two chunks of data…

	The server gets both chunks of data.
	The server exits after the first chunk of data is received.
	The server gets only one chunk of data.
	The proxy closes after handling the first chunk of data.
	All the above.
	None of the above.

Consider code snippet 4:

```
var net = require('net');

var LOCAL_PORT  = 8000;
var LOCAL_IP   = '127.0.0.1';
var REMOTE_PORT = 80;
var REMOTE_IP = '158.42.156.2';

var server = net.createServer(function (socket) {
    console.log("Client connected");
    var serviceSocket = new net.Socket();
    serviceSocket.connect(parseInt(REMOTE_PORT), REMOTE_IP, function (){
       socket.on('data', function (msg) {
           serviceSocket.write(msg);
       });
       serviceSocket.on('data', function (data) {
           socket.write(data);
       });
    });
}).listen(LOCAL_PORT, LOCAL_IP);
console.log("TCP server accepting connection on port: " + LOCAL_PORT);
```

Let P1 run this code instead of snippet 3, being the rest the same as for snippet 3, answer the following 3 questions.

14. When P2 connects to local port 8000, P1 …

	… connects to REMOTE_IP.
	… prints ‘`Client connected`’ to the console.
	… creates at most one connection to REMOTE_IP per client connection.
	… waits for data from REMOTE_IP as soon as it establishes connection with REMOTE_IP.
	All the above.
	None of the above.

15. Let P2 send two large chunks of data in sequence, D1 and then D2, through its connection to P1.
Then, in the absence of failures …

	D2 always arrives to REMOTE_IP after D1.
	D1 always arrives to REMOTE_IP after D2.
	Only D1 is relayed to REMOTE_IP.
	Only D2 is relayed to REMOTE_IP.
	All the above.
	None of the above.

16. Assume the server at REMOTE_IP can only handle one connection at a time.
Then, if P2 sends two chunks of data …

	The server gets both chunks of data.
	The server exits after the first chunk of data is received.
	The server gets only one chunk of data.
	The proxy closes after handling the first chunk of data.
	All the above.
	None of the above.

17. Both snippet 3 and 4 …

	Allow P1 to close the connection to REMOTE_IP when its client closes the connection.
	Allow P1 to close the connection with its clients when REMOTE_IP closes its connection.
	Need to be modified to capture the `'end'` event on `serviceSocket` and `socket` to properly handle closing the connections.
	Cannot be modified to properly handle connection closing.
	All the above.
	None of the above.

Consider code snippet 5:

```
var net = require('net');

var COMMAND_PORT  = 8000;
var LOCAL_IP  = '127.0.0.1';
var BASEPORT  = COMMAND_PORT + 1;
var remotes = []

function Remote(host, port, localPort) {
    this.targetHost = host;
    this.targetPort = port;
    this.server = net.createServer(function (socket)
        WHO.handleClientConnection(socket);
    });
    if (localPort) {
        this.server.listen(localPort, LOCAL_IP);
    }
}

Remote.prototype.handleClientConnection = function (socket) {
    var serviceSocket = new net.Socket();
    serviceSocket.connect(X_SERVER_PORT, X_SERVER_IP, function (){
       socket.on('data', function (msg) {
           serviceSocket.write(msg);
       });
       serviceSocket.on('data', function (data) {
           socket.write(data);
       });
       X_WHAT.on(X_EVENT, function () {
           serviceSocket.end();
       });
       X_WHICH.on(X_EVENT, function () {
           WHAT.end();
       });
    });
}

Remote.prototype.start = function (localPort) {
   this.server.listen(localPort, LOCAL_IP);
}

```

18. **In snippet 5, the variable** `WHO`

	Must be declared within the scope of the `Remote` function.
	Must point to the object being constructed.
	Cannot be null.
	Cannot be undefined.
	All the above.
	None of the above.

19. **In snippet 5,** `X_WHAT` **should be substituted by...**

	The client connection socket.
	The server connection socket.
	The Remote object.
	The server object.
	All the above.
	None of the above.

20. **In snippet 5,** `X_SERVER_PORT` **and** `X_SERVER_IP` **...**

	Refer to attributes `targetHost` and `targetPort` of the Remote object.
	Should be substituted by `this.targetHost` and `this.targetPort`, respectively.
	Can be the same for different Remote objects.
	Cannot be null or undefined.
	All the above.
	None of the above.