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`"

A	Brings all commits from the remote identified as "`origin`" to our local repository False. The "push" action tries to do the reverse transfer: from the local repository to the "origin" remote.
B	Deletes the current working directory False. To delete the current working copy is a dangerous action. That is not the aim of "push".
C	Always copies the local commits into the remote repository False. It is not guaranteed that such task is always done. That is the aim of "push" but in some cases it does not work as expected. If other users have successfully pushed other concurrent commits, our "push" attempt will fail. Git will tell us that we need a "pull" action in order to obtain those concurrent commits. Once those commits are merged with ours, we may try again our "push".
[✓ D]	When successful, sets the remote master branch pointing to the same commit as the local master branch True. In case of a successful completion, besides transferring our commits to the remote repository, the master branches of both repositories will point to the same commit.
E	All the above.
F	None of the above.

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

A	Are equivalent False. They are not equivalent. The "pull" action is a "fetch" followed by a "merge".
[✓ B]	Bring all commits of a remote repository to the local repository True. That is the result of a "fetch" action and is completed in all "pull" and "fetch" attempts.
C	Set the local master branch pointing to the same commit as the remote master branch. False. The "pull" action applies a "merge" once the commits from the remote master have been fetched. As a result of this, another commit is inserted in the local master branch. So, the local master branch doesn't point to the same commit as the remote master branch.
D	Modify the working directory. False. The working copy is only updated by "git pull". The "fetch" action doesn't modify that working copy.
E	All the above.
F	None of the above.

NOTE: Question 2 hasn't been considered for computing the marks since in some lab groups the "fetch" action was not explained.

3. Gitlab is…

A	A concrete server within the UPV
B	The name of a repository
C	A way of committing changes to a repository
D	An experimental version of Git
E	All the above.
[✓ F]	None of the above. As it is explained in the Lab1 bulletin, GitLab is a complementary system for Git. Basically, it provides web access to Git repositories and to the main Git actions. As a consequence of this, all the previous sentences are false.

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

A	Every developer in the team must have a private remote repository
B	There is a canonical repository that every member of team can read, different from any of the private remotes.
C	Only the owner can write to her private remote repository
D	Only one member of the team should write to the canonical repository
E	All the above. All the previous sentences are true. Each developer has a remote repository with a private accessibility flag (although its owner sets a Reporter role to the remaining team members, and this allows reading). The canonical repository has the characteristics stated in sentence B. Each private remote repository can be written only by its single owner, as stated in sentence C. The team leader is the unique member that should write to the canonical repository.
F	None of the above.

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

A	Should be accessible only by its owner No. All the other team members may read such repository.
B	Should be writeable by its owner True. The owner is the unique team member with write permissions on it.
C	Should be writeable by the integration manager No. The integration may read its contents, but she cannot write to it.
D	Should be writeable by the teacher No. Teachers may only read its contents.
E	All the above.
F	None of the above.

6. A local Git repository…

A	Must be associated to a remote Git repository False. In general, without considering the requirements seen in the practices, we can create local Git repositories without associating them to other (remote) repositories. Git is also valid for storing the history of versions (commits) in a given project being managed by a single user. In that case, no remote repository is needed.
B	Can be associated to only one remote repository False. The local repository being used in the integration step is associated to all the public repositories of all the team members, and to the canonical public repository. For instance, in a team of four members, as suggested in the labs, this means that such repository should be associated to five remote repositories.
C	Can be cloned onto another local repository. True. This is allowed by the "git clone" command.
D	Cannot be modified False. Git repositories should accept new commits. That is its main function: to store the evolution of a given project, maintaining the sequences followed by all the branches of commits that have happened in that project.
E	All the above.
F	None of the above.

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

A	"git init; touch README; git commit" True. The "git init" command, without arguments, creates an empty local repository in the directory where such command is executed. Assuming that we were using Git in a UNIX system, the semicolon character is used by its shell as a command separator. So, the next command creates an empty README file in the working copy of the repository just created. The last command is a "git commit" although it has no effect. Note that there is no file in the repository index, since we haven't used "git add" for placing the README file in such index. Thus, "git commit" fails and the repository remains empty.
B	"git clone git:init" False. "git clone" clones an existing repository, but it should be followed by a correct specification of the repository that must be cloned. Such specification may be a URL (for remote repositories) or a pathname (for local repositories). In this case, none of those alternatives has been followed. This implies that this command line doesn't work. It generates an error and it doesn't create any local repository.
C	"git init origin master" False. "git init" doesn't expect two additional arguments. This generates an error and that command doesn't work. The "origin" and "master" argument values might be needed in the "git pull" or "git push" commands when we have associated multiple remote repositories to our local one. However, they aren't needed in the "git init" command.
D	"git init -u origin master" False. The reason has been explained in the previous element.
E	All the above.
F	None of the above. Since the "git commit" command in part A fails, and this might be interpreted as an "erroneous" command line, this F alternative has also been accepted as a correct option in this question.

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…

A	Nothing changes at R False. One of the commits has been successfully pushed onto R. Thus, it has changed.
B	The master branch at R will point to C1 False. With the description given above, we cannot know which "git push" attempt has been successful. One of them has worked correctly, but we cannot assure that it has been D1's push.
C	The master branch at R will point to C2 False. With the description given above, we cannot know which "git push" attempt has been successful. One of them has worked correctly, but we cannot assure that it has been D2's push.
[✓ D]	One or more developers get an error when executing "`git push`" True. The second "git push" attempt fails. It finds a remote repository that doesn't store an identical sequence of commits to that in the push requester's repository. As a result, git rejects such "git push" attempt, recommending a "git pull" before trying a new push.
E	All the above.
F	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 ...

A	P1 sends an answer message to P2 after receiving the request message. False. P1 sends the "World" message as soon as it connects to P2 and closes such connection as soon as that message is sent.
B	When P1 starts to listen it outputs the message '`server connected`'. False. It prints the message "server bound" at that time.
[✓ C]	P2 may receive the message "World" before sending any information to P1. True. That message is sent by P1 as soon as P2 sets a connection with it.
D	P1 does not necessarily disconnect when P2 closes its socket. False. If any of the communication agents closes the socket, that connection is automatically closed. So, if P2 closes the socket, P1 is disconnected immediately.
E	All the above.
F	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 ...

A	P1 outputs a console message when P2 closes its socket. No. Snippet 2 has no listener function for the "end" event.
B	P1 can connect to P2 even though P1 and P2 are on different hosts. False. P1 only specifies a port in its connection attempt. So, it is connecting to that port on the local machine. P2 should be placed in the same computer to accept the connection from P1.
C	If P2 is not running, the message in variable '`text`' waits within the output buffer until P2 runs. False. If the server isn't listening for connections yet, the connection attempt from P1 generates an exception and P1 aborts. In the "net" module there is no asynchronous connection management.
D	We could invoke P1 from the command line like this `$ node client 127.0.0.1 8000 hello` False. No IP address is expected from the command line. Recall that process.argv[0] is "node", process.argv[1] is the name of the JavaScript file being run. In the programme, we observe that process.argv[2] is a server port number and that process.argv[3] is the string to be sent.
E	All the above.
[✓ F]	None of the above.

Consider code snippet 3:

```
15:  var net = require('net');
16:
17:  var LOCAL_PORT  = 8000;
18:  var LOCAL_IP  = '127.0.0.1';
19:  var REMOTE_PORT = 80;
20:  var REMOTE_IP = '158.42.156.2';
21:
22:  var server = net.createServer(function (socket) {
23:      socket.on('data', function (msg) {
24:         var serviceSocket = new net.Socket();
25:         serviceSocket.connect(parseInt(REMOTE_PORT), REMOTE_IP, function (){
26:             serviceSocket.write(msg);
27:         });
28:         serviceSocket.on('data', function (data) {
29:             socket.write(data);
30:         });
31:         console.log("Client connected");
32:      });
33:  }).listen(LOCAL_PORT, LOCAL_IP);
34:  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 …

A	… connects to REMOTE_IP. No. The connection handler (i.e., the callback used as the unique argument of the createServer() call) does not try to connect to that address. Instead of that, it is only installing a listener for “data” events. The code in that listener will be only executed when P1 receives data from that connection with P2.
B	… prints ‘`Client connected`’ to the console. No. The reason has been explained in part A.
C	… starts listening for data from REMOTE_IP. No. The reason has been explained in part A.
D	… stops accepting new connections. No. The reason has been explained in part A.
E	All the above.
F	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, …

A	D2 always arrives to REMOTE_IP after D1
B	D1 always arrives to REMOTE_IP after D2
C	Only D1 is relayed to REMOTE_IP
D	Only D2 is relayed to REMOTE_IP
E	All the above.
F	None of the above. Once a message is received from the P2 connection, P1 sets a new connection to REMOTE_IP and forwards such received message to REMOTE_IP. Since a different connection is settled for each received message, the order of message reception in REMOTE_IP depends on what has happened in each connection. We cannot ensure a concrete arrival order. Because of this, sentences A and B are false. Since each message is relayed onto its own connection, both C and D are also false.

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

A	The server gets both chunks of data. False. Our proxy (i.e., P1) has tried to set a second connection for D2, but the server doesn't accept such second connection. So, the second connection will not be settled (it will fail and D2 will be dropped) and a single chunk of data is delivered to the server.
B	The server exits after the first chunk of data is received. False. The description given in this question doesn't say that the server exits in that case. Instead of this, it seems that P1 might exit trying to connect.
[✓ C]	The server gets only one chunk of data. True. This is the consequence of what has been explained in part A.
D	The proxy closes after handling the first chunk of data. False. It doesn't close the connections once the first message has been correctly handled.
E	All the above.
F	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 …

A	… connects to REMOTE_IP. Yes. Line 11 implements such connection attempt. That line is directly placed in the callback being used for handling connections from external processes. P1 is listening to new connections on port 8000, as shown in line 19.
B	… prints '`Client connected`' to the console. Yes. This is done in line 9, in the connection handler.
C	… creates at most one connection to REMOTE_IP per client connection. Yes. As explained in part A, that is the responsibility of the code presented in line 11.
D	… waits for data from REMOTE_IP as soon as it establishes connection with REMOTE_IP. Yes. This is done in the "data" event listener shown in lines 15-17. Note that such listener is set on the "serviceSocket" object used for connecting with REMOTE_IP.
[✓ E]	All the above.
F	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 …

[✓ A]	D2 always arrives to REMOTE_IP after D1. Yes, since there is a single connection between P1 and REMOTE_IP for managing all messages sent by P2. Such connection guarantees a FIFO order and both messages are correctly forwarded.
B	D1 always arrives to REMOTE_IP after D2. False. See the explanation in sentence A.
C	Only D1 is relayed to REMOTE_IP. False. See the explanation in sentence A.
D	Only D2 is relayed to REMOTE_IP. False. See the explanation in sentence A.
E	All the above.
F	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 …

[✓ A]	The server gets both chunks of data. True. As explained in question 15, both messages are sent through the same connection. P2 correctly manages both chunks of data. They are received without problems by the server at REMOTE_IP.
B	The server exits after the first chunk of data is received. False. See the explanation in sentence A.
C	The server gets only one chunk of data. False. See the explanation in sentence A.
D	The proxy closes after handling the first chunk of data. False. See the explanation in sentence A.
E	All the above.
F	None of the above.

17. Both snippet 3 and 4 …

A	Allow P1 to close the connection to REMOTE_IP when its client closes the connection. False. There is no management of the "end" events (that are generated when a connection is closed) in both programmes. So, there is no management for close-connection events.
B	Allow P1 to close the connection with its clients when REMOTE_IP closes its connection. False. There is no management of the "end" events (that are generated when a connection is closed) in both programmes. So, there is no management for close-connection events.
[✓ C]	Need to be modified to capture the 'end' event on serviceSocket and socket to properly handle closing the connections. True. There is no management of the "end" events (that are generated when a connection is closed) in both programmes. So, there is no management for close-connection events.
D	Cannot be modified to properly handle connection closing. False. A proper connection management can be easily implemented adding listeners for the "end" events.
E	All the above.
F	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`

A	Must be declared within the scope of the `Remote` function.
B	Must point to the object being constructed.
C	Cannot be null.
D	Cannot be undefined.
[✓ E]	All the above. The word WHO being used in that snippet should be replaced with the "this" string. As such, it makes true all previous sentences: it is in the scope of the Remote constructor, it represents the object being constructed, it isn't null and it isn't undefined.
F	None of the above.

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

[✓ A]	The client connection socket. True. It should be replaced by the “socket” string. That variable holds the client connection socket.
B	The server connection socket. False. It should be replaced by the “socket” string. That variable holds the client connection socket.
C	The Remote object. False. It should be replaced by the “socket” string. That variable holds the client connection socket.
D	The server object. False. It should be replaced by the “socket” string. That variable holds the client connection socket.
E	All the above.
F	None of the above.

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

A	Refer to attributes `targetHost` and `targetPort` of the Remote object.
B	Should be substituted by `this.targetHost` and `this.targetPort`, respectively.
C	Can be the same for different Remote objects.
D	Cannot be null or undefined.
[✓ E]	All the above. All previous sentences have the same implications. The X_SERVER_PORT and X_SERVER_IP correspond to the this.targetPort and this.targetHost, respectively. Different Remote objects may place the same values in those attributes. JavaScript doesn’t impose any constraint on this. They cannot be null or undefined since with those values no connection could be set.
F	None of the above.