L'examen consta de 20 qüestions d'opció múltiple. En cada qüestió hi ha una única resposta correcta. Les respostes han de proporcionar-se en una fulla SEPARADA que s'ha facilitat al costat d'aquest enunciat.

Totes les qüestions tenen el mateix valor. Si es responen correctament, aporten 0.5 punts a la nota obtinguda. Si la resposta és negativa, l'aportació és negativa i equivalent a 1/5 del valor correcte; és a dir, -0.1 punts. En cas de dubte es recomana deixar la resposta en blanc.

La durada d'aquesta part de l'examen és 1 hora.

1. L'ordre "`git push origin master`"

A	Porta tots els commits del dipòsit remot "origin" al nostre dipòsit local. Fals. L'acció "push" realitza la transferència oposada: des del dipòsit local al remot "origin".
B	Esborra el directori de treball actual. Fals. Esborrar la còpia de treball local és una acció perillosa. Això no és l'objectiu de "push".
C	Sempre copia els commit locals en el dipòsit remot. Fals. No es podrà garantir que sempre ocórrega això. És l'objectiu de "push" però en alguns casos no aconsegueix realitzar-ho. Si altres usuaris han fet un "push" previ dels seus propis commits, el nostre intent de "push" fallarà. En aquest cas, Git es queixarà i ens recomanarà fer prèviament un "pull" per a portar i integrar els commits pujats per altres usuaris. Una vegada hàgem completat aquestes accions, podrem reintentar el "push".
D	Quan té èxit, deixa la branca del dipòsit remot master apuntant al mateix commit que la branca master local. Vertader. Quan té èxit, a més de transferir els nostres commits al dipòsit remot, les branques "master" de tots dos dipòsits apuntaran al mateix commit.
E	Totes les anteriors.
F	Cap de les anteriors.

2. Les dos ordres "`git pull`" i "`git fetch`"

A	Són equivalents. Fals. No ho són. L'acció "pull" fa el mateix que la seqüència "fetch" + "merge".
B	Porten tots els commits del dipòsit remot al dipòsit local. Cert. Aquest és l'efecte de l'acció "fetch" i es duu a terme tant en realitzar un "pull" com en realitzar un "fetch".
C	Deixen la branca master local apuntant al mateix commit que la branca master remota. Fals. Un "pull" aplica un "merge" després d'haver transferit tots els commits des de la branca "master" del dipòsit remot. Com a resultat d'això, un altre commit s'insereix en la branca "master" local. Per tant, la branca master local no apunta al mateix commit que la branca master remota.
D	Modifiquen el directori de treball. Fals. El directori (o còpia) de treball només arriba a modificar-se en fer un "git pull". L'acció "fetch" no modifica el directori de treball.
E	Totes les anteriors.
F	Cap de les anteriors.

NOTA: La qüestió 2 no ha sigut considerada per a calcular les notes ja que en alguns grups de laboratori no es va explicar l'acció "fetch".

3. GitLab és…

A	Un servidor concret dins de la UPV.
B	El nom d'un dipòsit.
C	Una forma de confirmar (*commit*) canvis en un dipòsit.
D	Una versió experimental de Git.
E	Totes les anteriors.
F	Cap de les anteriors. En el butlletí de la primera pràctica s'explica que GitLab és un sistema que complementa a Git. Bàsicament, GitLab facilita accés web als dipòsits Git i permet utilitzar les principals accions de Git des de la interfície web. Per tant, totes les afirmacions donades en els apartats anteriors són falses.

4. En el flux de treball per a la fase de desenvolupament que s'ha sol·licitat utilitzar en GitLab...

A	Cada desenvolupador de l'equip ha de tenir un dipòsit remot privat.
B	Hi ha un dipòsit canònic que cada membre de l'equip pot llegir i que és diferent dels dipòsits remots privats.
C	Només el propietari pot escriure en el seu dipòsit remot privat.
D	Només un membre de l'equip ha d'escriure en el dipòsit canònic.
E	Totes les anteriors. Tots els apartats anteriors són certs. Cada desenvolupador té un dipòsit remot amb el *flag* d'accessibilitat privada activat (però el seu propietari també haurà fixat un rol "Reporter" als altres membres del seu equip i això els permetrà accedir en mode lectura). El dipòsit canònic té les característiques enunciades en l'apartat B. Cada dipòsit remot privat només pot ser escrit pel seu propietari, tal com afirma l'apartat C. El líder de l'equip és l'únic membre que pot escriure en el dipòsit canònic.
F	Cap de les anteriors.

5. En un equip TSR, cada dipòsit privat remot...

A	Ha de ser accessible només pel seu propietari. No. Tots els membres de l'equip han de poder llegir els dipòsits remots.
B	Ha de ser accessible en mode escriptura pel seu propietari. Cert. El propietari és l'únic membre de l'equip amb permisos d'escriptura sobre aquest tipus de dipòsit.
C	Ha de ser accessible en mode escriptura per l'integrador de l'equip. No. L'integrador ha de poder llegir el seu contingut, però no modificar-lo. Les modificacions que decidisca aprovar l'integrador han de deixar-se en el dipòsit canònic. Des d'allí ja les prendrà el propietari per a aplicar-les al seu dipòsit remot privat.
D	Ha de ser accessible en mode escriptura pel líder de l'equip. No. La raó és la mateixa que en l'apartat C.
E	Totes les anteriors.
F	Cap de les anteriors.

6. Un dipòsit Git local…

A	Ha d'estar associat a un dipòsit Git remot. Fals. En general, sense considerar els requisits recomanats en pràctiques, podem crear dipòsits Git locals sense associar-los a altres dipòsits (remots). Git també pot usar-se per a mantenir un historial de versions (commits) en un projecte determinat que estiga sent gestionat per un únic usuari. En tal cas, no es necessita cap dipòsit remot.
B	Només pot associar-se a un únic dipòsit remot. Fals. El dipòsit local utilitzat pel gestor d'integració està associat al dipòsit canònic i a tots els dipòsits remots dels membres de l'equip. Per exemple, en un equip de quatre desenvolupadors, com el suggerit en les pràctiques, això significa que aquest dipòsit estarà associat a cinc dipòsits remots diferents.
C	Pot clonar-se sobre un altre dipòsit local. Cert. Això pot fer-se amb l'ordre "git clone", especificant el nom de ruta del directori/dipòsit a clonar.
D	No pot ser modificat. Fals. Els dipòsits Git han d'acceptar nous "commits". Aquesta és la seua funció principal: emmagatzemar l'evolució d'un projecte determinat, mantenint les seqüències de commits seguides per totes les branques que haja pogut haver-hi en el projecte.
E	Totes les anteriors.
F	Cap de les anteriors.

7. Seleccione l'ordre que crea un dipòsit Git local buit...

A	**"**`git init; touch README; git commit`**"** Cert. L'ordre "git init", sense arguments, crea un dipòsit local buit en el directori on s'utilitze aquesta ordre. Assumint que estem utilitzant Git en un sistema UNIX, el caràcter punt i coma s'utilitza en l'intèrpret d'ordres per a separar ordres consecutives. Per tant, la següent ordre crea un fitxer README buit en la còpia de treball del dipòsit que acabem de crear. L'última ordre és un "git commit" que no tindrà cap efecte. Observe's que no hi ha cap fitxer en l'índex del nostre dipòsit, ja què no hem fet cap "git add" per a incloure README en l'índex. Així, "git commit" falla i el dipòsit roman buit.
B	**"**`git clone git:init`**"** Fals. "git clone" permet clonar un dipòsit existent, però per a fer això hem d'especificar correctament el dipòsit a clonar. Aquesta especificació pot ser un URL (per a dipòsits remots) o un nom de ruta (per a dipòsits locals). En aquest exemple, cap d'aquestes alternatives ha sigut utilitzada. Això implica que l'ordre no funcionarà. Genera un error i no crea un dipòsit local.
C	**"**`git init origin master`**"** Fals. L'ordre "git init" no espera dos arguments. Aquest intent genera un error i l'ordre no funciona. Els arguments "origin" i "master" poden ser necessaris en un "git pull" o en un "git push" quan tinguem múltiples dipòsits remots associats al nostre dipòsit local. En aquests casos "origin" és un àlies per al dipòsit i "master" és el nom de la branca sobre la qual es vol actuar. L'ordre "git init" no necessita aquests arguments.
D	**"**`git init -o origin master`**"** Fals. La raó ja s'ha explicat en l'apartat anterior.

E	Totes les anteriors.
F	Cap de les anteriors. Ja que l'ordre "git commit" provoca una fallada en l'apartat A i això podria interpretar-se com una línia d'ordres errònia... aquest apartat F també ha sigut considerat una opció acceptable en aquesta qüestió.

8. Assumisca la següent situació: El dipòsit R (la URL de la qual is R_{url}) pot ser accedit per dos desenvolupadors D1 i D2. La branca "master" de R apunta a un commit C que conté el fitxer "`foo.js`", amb una sola línia, "`var fs = require( 'fs' )`". Cada desenvolupador executa "`git clone` R_{url}`; cd R;`" en el seu ordinador. D1 edita "`foo.js`" i afig la línia "`fs.readFileSync( 'myfile' )`". D2 també edita "`foo.js`" però afig la línia "`fs.readFileSync( 'nofile' )`". Tots dos realitzen el commit dels seus canvis, en diferents moments, obtenint els commits C1 i C2, respectivament. Després, ells executen "`git push`".
Podem afirmar que…

A	No canvia res en R . Fals. Un dels commits ha sigut correctament enviat i aplicat sobre R. Per tant, R ha canviat.
B	La branca "master" de R apuntarà a C1. Fals. Amb la descripció donada en l'enunciat no podem saber quin "git push" ha sigut acceptat. Un dels dos ho haurà sigut (el primer a ser rebut), però no podem assegurar que haja sigut el de D1.
C	La branca "master" de R apuntarà a C2. Fals. Amb la descripció donada en l'enunciat no podem saber quin "git push" ha sigut acceptat. Un dels dos ho haurà sigut (el primer a ser rebut), però no podem assegurar que haja sigut el de D2.
D	Un o més desenvolupadors obtindran un error quan executen "`git push`". Cert. El segon "git push" fallarà. Troba un dipòsit remot que no emmagatzema la mateixa seqüència de commits mantinguda en el dipòsit local des del qual es va sol·licitar el "push". Per tant, Git rebutja el "git push" recomanant un "git pull" abans de tornar a intentar-ho.
E	Totes les anteriors.
F	Cap de les anteriors.

Considere el següent fragment 1:

```
01:   var net = require('net');
02:   var server = net.createServer(  function(c) {
03:      console.log('server connected');
04:      c.write('World');
05:      c.end();
06:   });
07:
08:   server.listen(8000, function() {
09:     console.log('server bound');
10:   });
11:
```

9. Assumisca que el procés P1 executa el codi del fragment 1. Si assumim que P2 és un altre procés iniciat en el mateix ordinador que P1 i que P2 es connecta al port 8000, llavors…

A	P1 envia la resposta a P2 després de rebre el missatge de petició. Fals. P1 envia el missatge "World" quan P2 es connecta i tanca aquesta connexió després d'enviar aquest missatge. No espera cap missatge de petició.
B	Quan P1 es posa a escoltar imprimeix el missatge 'server connected'. Fals. Escriu el missatge "server bound" quan això ocorre.
C	P2 podria rebre el missatge 'World' abans d'enviar cap informació a P1. Cert. Aquest missatge és enviat per P1 quan P2 estableix la connexió amb ell.
D	Si P2 tanca el seu socket no implica que P1 es desconnecte. Fals. Si un procés tanca un socket, la connexió també es tanca. Per tant, si P2 tanca el socket, P1 queda desconnectat.
E	Totes les anteriors.
F	Cap de les anteriors.

Considere el següent fragment 2:

```
12:  var net = require('net');
13:
14:  var server_port = process.argv[2];
15:  var text        = process.argv[3].toString();
16:  var client      = net.connect({port: parseInt(server_port}, function() {
17:     console.log('client connected');
18:     // This will be echoed by the server.
19:     client.write(text);
20:  });
21:  client.on('data', function(data) {
22:    console.log(data.toString());
23:    client.end();
24:  });
25:
```

10. Supose que el procés P1 executa aquest fragment 2. Siga P2 un procés servidor que escolta peticions en el port 8000. Llavors …

A	P1 trau un missatge per pantalla quan P2 tanca el socket. No. El fragment 2 no té cap "listener" per a l'esdeveniment "end".
B	La connexió es produirà encara que P2 no s'execute en la mateixa màquina que P1. No. P1 solament especifica un nombre de port quan estableix la connexió. Per tant, està connectant-se a aquest port en l'ordinador local. P2 ha d'estar situat en el mateix ordinador per a acceptar la connexió amb P1.
C	Si el servidor no està en marxa el missatge contingut en la variable 'text' quedarà a l'espera en el buffer d'eixida fins que el servidor estiga en marxa. Fals. Si el servidor no espera connexions (mitjançant listen()), l'intent de connexió de P1 generarà una excepció i P1 avortarà. El mòdul "net" no ofereix una gestió asincròncia de connexions. Aquesta funcionalitat l'oferia ZeroMQ.
D	Es podria invocar P1 des de la línia d'ordres de Linux com: $ node client 127.0.0.1 8000 hello Fals. No s'espera cap adreça IP des de la línia d'ordres. Ha de recordar-se que process.argv[0] serà la cadena "node" i process.argv[1] és el nom del fitxer JavaScript que estem executant. En aquest exemple, process.argv[2] és el nombre de port del servidor i process.argv[3] és la cadena a enviar.
E	Totes les anteriors.
F	Cap de les anteriors.

Considere el fragment 3 següent:

```
26:   var net = require('net');
27:
28:   var LOCAL_PORT  = 8000;
29:   var LOCAL_IP  = '127.0.0.1';
30:   var REMOTE_PORT = 80;
31:   var REMOTE_IP = '158.42.156.2';
32:
33:   var server = net.createServer(function (socket) {
34:       socket.on('data', function (msg) {
35:          var serviceSocket = new net.Socket();
36:          serviceSocket.connect(parseInt(REMOTE_PORT), REMOTE_IP, function (){
37:              serviceSocket.write(msg);
38:          });
39:          servicesocket.on('data', function (data) {
40:              socket.write(data);
41:          });
42:          console.log("Client connected");
43:       });
44:   }).listen(LOCAL_PORT, LOCAL_IP);
45:   console.log("TCP server accepting connection on port: " + LOCAL_PORT);
```

Assumisca que aquest codi s'executa per a generar el procés P1. Les dades dels clients connectats al port 8000 s'assumeix que arriben en blocs a través de la connexió que aquests estableixen. Cada bloc genera un esdeveniment "data" en el socket que manté la connexió.

Assumisca un procés P2 iniciat després de P1 en l'ordinador de P1 i que també es connecta al port local 8000. Responga les tres qüestions següents:

11. Quan P2 es connecta al port local 8000, P1 …

A	… es connecta a REMOTE_IP. Fals. La funció que gestiona la connexió (és a dir, el *callback* utilitzat com a únic argument de la crida createServer()) no tracta de connectar-se a aquesta adreça. En lloc d'això, només instal·la un "listener" per a esdeveniments "data". El codi del "listener" només s'executa quan P1 rep missatges des de la connexió amb P1.
B	… imprimeix '`Client connected`' en la consola. Fals. Veure explicació de l'apartat A.
C	… comença a esperar dades enviades des de REMOTE_IP. Fals. Veure explicació de l'apartat A.
D	… deixa d'acceptar noves connexions. Fals. Veure explicació de l'apartat A.
E	Totes les anteriors.
F	Cap de les anteriors.

12. Assumim que P2 envia dos blocs grans de dades en seqüència, D1 i després D2, a través de la seua connexió amb P1.
Llavors, en absència de fallades,...

A	D2 sempre arriba a REMOTE_IP després de D1.
B	D1 sempre arriba a REMOTE_IP després de D2.
C	Només D1 es transmet a REMOTE_IP.
D	Només D2 es transmet a REMOTE_IP.
E	Totes les anteriors.
F	Cap de les anteriors. Quan reba un missatge des de la connexió amb P2, P1 establirà una nova connexió amb REMOTE_IP i reenviarà el missatge rebut cap a REMOTE_IP. Ja que s'estableix una connexió diferent per cada missatge rebut, l'ordre de recepció en REMOTE_IP dependrà del que ocórrega en cada connexió. No podem assegurar cap ordre d'arribada concret. Per això, les opcions A i B són falses. Com cada missatge és reenviat per la seua pròpia connexió, C i D també són falses.

13. Assumim que el servidor en REMOTE_IP només poguera manejar una connexió alhora.
Llavors, si P2 enviara dos blocs de dades...

A	El servidor obtindria dos blocs de dades. Fals. El nostre proxy (és a dir, P1) intentarà establir una segona connexió per a D2, però el servidor no l'acceptarà. Per tant, la segona connexió no arribarà a establir-se (fallarà i D2 es perdrà) i solament un dels blocs de dades serà lliurat al servidor.
B	El servidor acabaria després de rebre el primer bloc de dades. Fals. La descripció donada en l'enunciat no suggereix aquest comportament. En el seu lloc, sembla que vaja a ser P1 qui avorte en tractar de connectar-se per segona vegada.
C	El servidor rep un sol bloc de dades. Cert. És una conseqüència del que hem descrit en l'apartat A.
D	El proxy es tanca després de manejar el primer bloc de dades. Fals. No tanca les connexions ni acaba voluntàriament després d'haver manejat correctament el primer bloc de dades.
E	Totes les anteriors.
F	Cap de les anteriors.

Considere el fragment 4 següent:

```
01:  var net = require('net');
02:
03:  var LOCAL_PORT  = 8000;
04:  var LOCAL_IP  = '127.0.0.1';
05:  var REMOTE_PORT = 80;
06:  var REMOTE_IP = '158.42.156.2';
07:
08:  var server = net.createServer(function (socket) {
09:      console.log("Client connected");
10:      var serviceSocket = new net.Socket();
11:      serviceSocket.connect(parseInt(REMOTE_PORT), REMOTE_IP, function (){
12:         socket.on('data', function (msg) {
13:             serviceSocket.write(msg);
14:         });
15:         servicesocket.on('data', function (data) {
16:             socket.write(data);
17:         });
18:      });
19:  }).listen(LOCAL_PORT, LOCAL_IP);
20:  console.log("TCP server accepting connection on port: " + LOCAL_PORT);
```

Assumisca que P1 executa aquest codi en comptes del fragment 3, sent la resta idèntica al descrit per al fragment 3. Conteste les tres qüestions següents:

14. Quan P2 es connecta al port local 8000, P1 …

A	… es connecta a REMOTE_IP. Cert. La línia 11 implanta aquest intent de connexió. Aquesta línia està situada en el *callback* utilitzat per a manejar connexions des d'altres processos. P1 està esperant noves connexions en el port 8000, com mostra la línia 19.
B	… imprimeix '`Client connected`' en la consola. Cert. Això ocorre en la línia 9, dins de la funció que gestiona les connexions.
C	… crea com a màxim una connexió a REMOTE_IP per cada connexió client. Cert. Tal com s'ha explicat en l'apartat A, aquesta és la responsabilitat del codi mostrat en la línia 11.
D	… espera dades des de REMOTE_IP tan aviat com establisca una connexió amb REMOTE_IP. Cert. Això es fa en la funció que gestiona els esdeveniments "data", mostrada en les línies 15 a 17. Observe's que aquesta funció s'ha associat a l'objecte "serviceSocket", utilitzat per a connectar amb REMOTE_IP.
E	Totes les anteriors.
F	Cap de les anteriors.

15. P2 envia dos blocs grans de dades en seqüència, D1 seguit per D2, a través de la seua connexió amb P1.
Llavors, en absència de fallades …

A	D2 sempre arriba a REMOTE_IP després de D1. Sí, ja què només hi ha una única connexió entre P1 i REMOTE_IP per a gestionar tots els missatges enviats per P2. Aquesta connexió garanteix ordre FIFO i tots dos missatges seran propagats correctament si no hi ha fallades.
B	D1 sempre arriba a REMOTE_IP després de D2. Fals. Vegeu la justificació de l'apartat A.
C	Solament D1 és transmès a REMOTE_IP. Fals. Vegeu la justificació de l'apartat A.
D	Solament D2 és transmès a REMOTE_IP. Fals. Vegeu la justificació de l'apartat A.
E	Totes les anteriors.
F	Cap de les anteriors.

16. Assumisca que el servidor en REMOTE_IP puga gestionar només una connexió alhora. Llavors, si P2 envia dos blocs de dades …

A	El servidor obté tots dos blocs de dades. Cert. Com s'ha explicat en la qüestió 15, tots dos missatges es propaguen per la mateixa connexió. P2 gestiona correctament tots dos blocs de dades. Són rebuts sense problemes pel servidor executat en REMOTE_IP.
B	El servidor acaba després de rebre el primer bloc de dades. Fals. Vegeu la justificació de l'apartat A.
C	El servidor solament rep un bloc de dades. Fals. Vegeu la justificació de l'apartat A.
D	El proxy es tanca després de gestionar el primer bloc de dades. Fals. Vegeu la justificació de l'apartat A.
E	Totes les anteriors.
F	Cap de les anteriors.

17. Tots dos fragments 3 i 4 …

A	Permeten a P1 tancar la connexió a REMOTE_IP quan el client tanca la connexió. Fals. No hi ha cap gestió per als esdeveniments "end" (que són els generats quan una connexió es tanca) en aquests dos fragments. Per tant, P1 no pot tancar la connexió en aquest cas.
B	Permeten a P1 tancar la connexió amb els seus clients quan REMOTE_IP tanque la seua connexió. Fals. No hi ha cap gestió per als esdeveniments "end" (que són els generats quan una connexió es tanca) en aquests dos fragments. Per tant, P1 no pot tancar la connexió en aquest cas.
C	Necessiten ser modificats per a capturar l'esdeveniment `'end'` en `serviceSocket` i `socket` per a gestionar adeqüadament el tancament de connexions. Vertader. No hi ha cap gestió per als esdeveniments "end" (que són els generats quan una connexió es tanca) en aquests dos fragments. Per tant, hem d'estendre'ls com suggereix aquest apartat.
D	No poden ser modificats per a gestionar adeqüadament el tancament de connexions. Fals. Una gestió adequada del tancament de connexions pot implantar-se fàcilment afegint funcions que actuen com a *listeners* dels esdeveniments "end".
E	Totes les anteriors.
F	Cap de les anteriors.

Considere el següent fragment 5:

```
01:   var net = require('net');
02:
03:   var COMMAND_PORT  = 8000;
04:   var LOCAL_IP  = '127.0.0.1';
05:   var BASEPORT  = COMMAND_PORT + 1;
06:   var remotes = []
07:
08:   function Remote(host, port, localPort) {
```

```
09:       this.targetHost = host;
10:       this.targetPort = port;
11:       this.server = net.createServer(function (socket)
12:           WHO.handleClientConnection(socket);
13:       });
14:       if (localPort) {
15:           this.server.listen(localPort, LOCAL_IP);
16:       }
17:   }
18:
19:   Remote.prototype.handleClientConnection = function (socket) {
20:       var serviceSocket = new net.Socket();
21:       serviceSocket.connect(X_SERVER_PORT, X_SERVER_IP, function (){
22:          socket.on('data', function (msg) {
23:              serviceSocket.write(msg);
24:          });
25:          servicesocket.on('data', function (data) {
26:              socket.write(data);
27:          });
28:          X_What.on(X_EVENT, function () {
29:              serviceSocket.end();
30:          });
31:          X_Which.on(X_EVENT, function () {
32:              WHAT.end();
33:          });
34:       });
35:   }
36:
37:   Remote.prototype.start = function (localPort) {
38:      this.server.listen(localPort, LOCAL_IP);
39:   }
40:
```

18. **En el fragment 5, la variable** `WHO`

A	Ha de declarar-se en l'àmbit de la funció `Remote`.
B	Ha d'apuntar a l'objecte que s'està construint.
C	No pot ser null.
D	No pot ser undefined.
E	Totes les anteriors. L'identificador WHO utilitzat en aquest fragment hauria de ser reemplaçat per la cadena `this`. Amb això tots els apartats anteriors són certs: està en l'àmbit del constructor de `Remote`, representa a l'objecte que estem construint, no pot ser `null` i no pot ser `undefined`.
F	Cap de les anteriors.

19. **En el fragment 5, `X_WHAT` ha de substituir-se per...**

A	El socket de la connexió client. Cert. Hauria de ser substituït per la cadena `socket`. Aquesta variable manté el socket amb la connexió establida amb el client.
B	El socket de la connexió del servidor. Fals. Vegeu la justificació del primer apartat.
C	L'objecte Remote. Fals. Vegeu la justificació del primer apartat.
D	L'objecte servidor. Fals. Vegeu la justificació del primer apartat.
E	Totes les anteriors.
F	Cap de les anteriors.

20. **En el fragment 5, `X_SERVER_PORT` i `X_SERVER_IP` ...**

A	Es refereixen als atributs `targetHost` i `targetPort` de l'objecte Remote.
B	Han de substituir-se per `this.targetHost` i `this.targetPort`, respectivament.
C	Poden ser idèntics per a diferents objectes Remote.
D	No poden ser null o undefined
E	Totes les anteriors. Tots els apartats anteriors impliquen el mateix: X_SERVER_PORT i X_SERVER_IP corresponen a `this.targetPort` i `this.targetHost`, respectivament. Diferents objectes Remote poden tenir els mateixos valors en aquests atributs. JavaScript no imposa cap restricció que ho impedisca. No poden ser `null` ni `undefined` ja què amb aquests valors no es podria establir cap connexió.
F	Cap de les anteriors.