El examen consta de 20 cuestiones de opción múltiple. En cada cuestión hay una sola respuesta correcta. Las respuestas deben proporcionarse en una hoja SEPARADA que se ha facilitado junto a este enunciado.

Todas las cuestiones tienen el mismo valor. Si se responden correctamente, aportan 0.5 puntos a la nota obtenida. Si la respuesta es negativa, la aportación es negativa y equivalente a 1/5 del valor correcto; es decir, -0.1 puntos. En caso de duda se recomienda dejar la respuesta en blanco.

La duración de esta parte del examen es 1 hora.

1. La orden "`git push origin master`"

A	Trae todos los commits del depósito remoto "`origin`" a nuestro depósito local. Falso. La acción "push" realiza la transferencia opuesta: desde el depósito local al remoto "origin".
B	Borra el directorio de trabajo actual. Falso. Borrar la copia de trabajo local es una acción peligrosa. Ese no es el objetivo de "push".
C	Siempre copia los commit locales en el depósito remoto. Falso. No se podrá garantizar que siempre ocurra eso. Es el objetivo de "push" pero en algunos casos no logra realizarla. Si otros usuarios han realizado un "push" previo de sus propios commits, nuestro intento de "push" fallará. En ese caso, Git se quejará y nos recomendará realizar previamente un "pull" para traer e integrar los commits subidos por otros usuarios. Una vez hayamos completado esas acciones, podremos reintentar el "push".
D	Cuando tiene éxito, deja la rama del depósito remoto master apuntando al mismo commit que la rama master local. Verdadero. Cuando tiene éxito, además de transferir nuestros commits al depósito remoto, las ramas "master" de ambos depósitos apuntarán al mismo commit.
E	Todas las anteriores.
F	Ninguna de las anteriores.

2. Las dos órdenes "`git pull`" y "`git fetch`"

A	Son equivalentes. Falso. No lo son. La acción "pull" hace lo mismo que la secuencia "fetch" + "merge".
B	Traen todos los commits del depósito remoto al depósito local. Cierto. Ese es el efecto de la acción "fetch" y se lleva a cabo tanto al realizar un "pull" como al realizar un "fetch".
C	Dejan la rama master local apuntando al mismo commit que la rama master remota. Falso. Un "pull" aplica un "merge" tras haber transferido todos los commits desde la rama "master" del depósito remoto. Como resultado de ello, otro commit se inserta en la rama "master" local. Por tanto, la rama master local no apunta al mismo commit que la rama master remota.
D	Modifican el directorio de trabajo. Falso. El directorio (o copia) de trabajo solo llega a modificarse al realizar un "git pull". La acción "fetch" no modifica el directorio de trabajo.
E	Todas las anteriores.
F	Ninguna de las anteriores.

NOTA: La cuestión 2 no ha sido considerada para calcular las notas ya que en algunos grupos de laboratorio no se explicó la acción "fetch".

3. GitLab es…

A	Un servidor concreto dentro de la UPV.
B	El nombre de un depósito.
C	Una forma de confirmar (*commit*) cambios en un depósito.
D	Una versión experimental de Git.
E	Todas las anteriores.
F	Ninguna de las anteriores. En el boletín de la primera práctica se explica que GitLab es un sistema que complementa a Git. Básicamente, GitLab facilita acceso web a los depósitos Git y permite utilizar las principales acciones de Git desde esa interfaz web. Por tanto, todas las afirmaciones dadas en los apartados anteriores son falsas.

4. En el flujo de trabajo para la fase de desarrollo que se ha solicitado utilizar en GitLab...

A	Cada desarrollador del equipo debe tener un depósito remoto privado.
B	Hay un depósito canónico que cada miembro del equipo puede leer y que es diferente de los depósitos remotos privados.
C	Solo el propietario puede escribir en su depósito remoto privado.
D	Solo un miembro del equipo debe escribir en el depósito canónico.
E	Todas las anteriores. Todos los apartados anteriores son ciertos. Cada desarrollador tiene un depósito remoto con el *flag* de accesibilidad privada activado (pero su propietario también habrá fijado un rol "Reporter" a los demás miembros de su equipo y eso les permitirá acceder en modo lectura). El depósito canónico tiene las características enunciadas en el apartado B. Cada depósito remoto privado solo puede ser escrito por su propietario, tal como afirma el apartado C. El líder del equipo es el único miembro que puede escribir en el depósito canónico.
F	Ninguna de las anteriores.

5. En un equipo TSR, cada depósito privado remoto...

A	Debe ser accesible solo por su propietario. No. Todos los miembros del equipo deben poder leer los depósitos remotos.
B	Debe ser accesible en modo escritura por su propietario. Cierto. El propietario es el único miembro del equipo con permisos de escritura sobre ese tipo de depósito.
C	Debe ser accesible en modo escritura por el integrador del equipo. No. El integrador debe poder leer su contenido, pero no modificarlo. Las modificaciones que decida aprobar el integrador deben dejarse en el depósito canónico. Desde allí ya las tomará el propietario para aplicarlas a su depósito remoto privado.
D	Debe ser accesible en modo escritura por el líder del equipo. No. La razón es la misma que en el apartado C.
E	Todas las anteriores.
F	Ninguna de las anteriores.

6. Un depósito Git local…

A	Debe estar asociado a un depósito Git remoto. Falso. En general, sin considerar los requisitos recomendados en prácticas, podemos crear depósitos Git locales sin asociarlos a otros depósitos (remotos). Git también puede usarse para mantener un historial de versiones (commits) en un proyecto determinado que está siendo gestionado por un único usuario. En tal caso, no se necesita ningún depósito remoto.
B	Solo puede asociarse a un único depósito remoto. Falso. El depósito local utilizado por el gestor de integración está asociado al depósito canónico y a todos los depósitos remotos de los miembros del equipo. Por ejemplo, en un equipo de cuatro desarrolladores, como el que se sugiere en las prácticas, esto significa que ese depósito estará asociado a cinco depósitos remotos diferentes.
C	Puede clonarse sobre otro depósito local. Cierto. Esto puede hacerse con la orden "git clone", especificando el nombre de ruta del directorio/depósito a clonar.
D	No puede ser modificado. Falso. Los depósitos Git deben aceptar nuevos "commits". Esa es su principal función: almacenar la evolución de un proyecto determinado, manteniendo las secuencias de commits seguidas por todas las ramas que haya podido haber en el proyecto.
E	Todas las anteriores.
F	Ninguna de las anteriores.

7. Seleccione la orden que crea un depósito Git local vacío...

A	**"**`git init; touch README; git commit`**"** Cierto. La orden "git init", sin argumentos, crea un depósito local vacío en el directorio donde se utilice esa orden. Asumiendo que estamos utilizando Git en un sistema UNIX, el carácter punto y coma se utiliza en el intérprete de órdenes para separar órdenes consecutivas. Por tanto, la siguiente orden crea un fichero README vacío en la copia de trabajo del depósito que acabamos de crear. La última orden es un "git commit" que no tendrá ningún efecto. Obsérvese que no hay ningún fichero en el índice de nuestro depósito, pues no hemos realizado ningún "git add" para incluir README en ese índice. Así, "git commit" falla y el depósito permanece vacío.
B	**"**`git clone git:init`**"** Falso. "git clone" permite clonar un depósito existente, pero para ello debemos especificar correctamente el depósito a clonar. Esa especificación puede ser un URL (para depósitos remotos) o un nombre de ruta (para depósitos locales). En este ejemplo, ninguna de esas alternativas ha sido utilizada. Esto implica que esa orden no funcionará. Genera un error y no crea un depósito local.
C	**"**`git init origin master`**"** Falso. La orden "git init" no espera dos argumentos. Este intento genera un error y la orden no funciona. Los argumentos "origin" y "master" pueden ser necesarios en un "git pull" o en un "git push" cuando tengamos múltiples depósitos remotos asociados a nuestro depósito local. En ese caso "origin" es un alias para el depósito y "master" es el nombre de la rama sobre la que se va a actuar. La orden "git init" no necesita esos argumentos.

D	**"**`git init -u origin master`**"** Falso. La razón ya se ha explicado en el apartado anterior.
E	Todas las anteriores.
F	Ninguna de las anteriores. Ya que la orden "git commit" provoca un fallo en el apartado A y eso podría interpretarse como una línea de órdenes errónea... este apartado F también ha sido considerado una opción aceptable en esta cuestión.

8. Asuma la siguiente situación: El depósito R (cuya URL es R_{url}) puede ser accedido por dos desarrolladores D1 y D2. La rama "master" de R apunta a un commit C que contiene el fichero "`foo.js`**", con una sola línea, "**`var fs = require('fs')`**". Cada desarrollador ejecuta "**`git clone` R_{url}`; cd R;`**" en su ordenador. D1 edita "**`foo.js`**" y añade la línea "**`fs.readFileSync('myfile')`**". D2 también edita "**`foo.js`**" pero añade la línea "**`fs.readFileSync('nofile')`**". Ambos realizan el commit de sus cambios, en distintos momentos, obteniendo los commits C1 y C2, respectivamente. Entonces ellos ejecutan "**`git push`**". Podemos afirmar que…**

A	Nada cambia en R. Falso. Uno de los commits ha sido correctamente enviado y aplicado sobre R. Por tanto, R ha cambiado.
B	La rama "master" de R apuntará a C1. Falso. Con la descripción dada en el enunciado no podemos saber qué "git push" ha sido aceptado. Uno de los dos lo habrá sido (el primero en ser recibido), pero no podemos asegurar que haya sido el de D1.
C	La rama "master" de R apuntará a C2. Falso. Con la descripción dada en el enunciado no podemos saber qué "git push" ha sido aceptado. Uno de los dos lo habrá sido (el primero en ser recibido), pero no podemos asegurar que haya sido el de D2.
D	Uno o más desarrolladores obtendrán un error cuando ejecuten **"**`git push`**".** Cierto. El segundo "git push" fallará. Encuentra un depósito remoto que no almacena la misma secuencia de commits mantenida en el depósito local desde el que se solicitó ese "push". Por tanto, Git rechaza ese "git push" recomendando un "git pull" antes de volver a intentarlo.
E	Todas las anteriores.
F	Ninguna de las anteriores.

Considere el siguiente fragmento 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. Asuma que el proceso P1 ejecuta el código del fragmento 1. Si asumimos que P2 es otro proceso iniciado en el mismo ordenador que P1 y que P2 se conecta al puerto 8000, entonces…

A	P1 envía la respuesta a P2 después de recibir el mensaje de petición. Falso. P1 envía el mensaje "World" cuando P2 se conecta y cierra esa conexión tras enviar ese mensaje. No espera ningún mensaje de petición.
B	Cuando P1 se pone en escucha imprime el mensaje 'server connected'. Falso. Escribe el mensaje "server bound" cuando eso sucede.
C	P2 podría recibir el mensaje 'World' antes de enviar ninguna información a P1. Cierto. Ese mensaje es enviado por P1 cuando P2 establece la conexión con él.
D	Si P2 cierra su socket no implica que P1 se desconecte. Falso. Si un proceso cierra un socket, esa conexión también se cierra. Por tanto, si P2 cierra el socket, P1 queda desconectado.
E	Todas las anteriores.
F	Ninguna de las anteriores.

Considere el siguiente fragmento 2:

```
12:   var net = require('net');
13:
14:   var server_port = process.argv[2];
15:   var texto       = 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(texto);
20:   });
21:   client.on('data', function(data) {
22:     console.log(data.toString());
23:     client.end();
24:   });
25:
```

10. Suponga que el proceso P1 ejecuta este fragmento 2. Sea P2 un proceso servidor que escucha peticiones en el puerto 8000. Entonces …

A	P1 saca un mensaje por pantalla cuando P2 cierra el socket. No. El fragmento 2 no tiene ningún "listener" para el evento "end".
B	La conexión se producirá aunque P2 no se ejecute en la misma máquina que P1. No. P1 solo especifica un número de puerto cuando establece la conexión. Por tanto, está conectándose a ese puerto en el ordenador local. P2 debe estar ubicado en el mismo ordenador para aceptar la conexión con P1.
C	Si el servidor no está en marcha el mensaje contenido en la variable 'texto' quedará a la espera en el buffer de salida hasta que el servidor esté en marcha. Falso. Si el servidor no espera conexiones (mediante listen()), el intento de conexión de P1 generará una excepción y P1 abortará. El módulo "net" no ofrece una gestión asíncrona de conexiones. Esa funcionalidad la ofrecía ZeroMQ.
D	Se podría invocar P1 desde la línea de órdenes de Linux como : $ node client 127.0.0.1 8000 hello Falso. No se espera ninguna dirección IP desde la línea de órdenes. Debe recordarse que process.argv[0] será la cadena "node" y process.argv[1] es el nombre del fichero JavaScript que estamos ejecutando. En este ejemplo, process.argv[2] es el número de puerto del servidor y process.argv[3] es la cadena a enviar.
E	Todas las anteriores.
F	Ninguna de las anteriores.

Considere el fragmento 3 siguiente:

```
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);
```

Asuma que este código se ejecuta para generar el proceso P1. Los datos de los clientes conectados al puerto 8000 se asume que llegan en bloques a través de la conexión que estos establecen. Cada bloque genera un evento "data" en el socket que mantiene la conexión.

Asuma un proceso P2 iniciado tras P1 en el ordenador de P1 y que también se conecta al puerto local 8000. Responda las tres cuestiones siguientes:

11. Cuando P2 se conecta al puerto local 8000, P1 …

A	… se conecta a REMOTE_IP. Falso. La función que gestiona esa conexión (esto es, el *callback* utilizado como único argumento de la llamada createServer()) no trata de conectarse a esa dirección. En lugar de ello, solo instala un "listener" para eventos "data". El código de ese "listener" solo llega a ejecutarse cuando P1 recibe mensajes desde la conexión con P1.
B	… imprime '`Client connected`' en la consola. Falso. Ver explicación del apartado A.
C	… empieza a esperar datos enviados desde REMOTE_IP. Falso. Ver explicación del apartado A.
D	… deja de aceptar nuevas conexiones. Falso. Ver explicación del apartado A.
E	Todas las anteriores.
F	Ninguna de las anteriores.

12. Asumamos que P2 envía dos bloques grandes de datos en secuencia, D1 y después D2, a través de su conexión con P1.
Entonces, en ausencia de fallos,...

A	D2 siempre llega a REMOTE_IP tras D1.
B	D1 siempre llega a REMOTE_IP tras D2.
C	Solo D1 se transmite a REMOTE_IP.
D	Solo D2 se transmite a REMOTE_IP.
E	Todas las anteriores.
F	Ninguna de las anteriores. Cuando reciba un mensaje desde la conexión con P2, P1 establecerá una nueva conexión con REMOTE_IP y reenviará ese mensaje recibido hacia REMOTE_IP. Ya que se establece una conexión diferente por cada mensaje recibido, el orden de recepción en REMOTE_IP dependerá de lo que ocurra en cada conexión. No podemos asegurar ningún orden de llegada concreto. Por ello, las opciones A y B son falsas. Como cada mensaje es reenviado por su propia conexión, tanto C como D también son falsas.

13. Asumamos que el servidor en REMOTE_IP solo pudiera manejar una conexión a la vez.
Entonces, si P2 enviase dos bloques de datos...

A	El servidor obtendría dos bloques de datos. Falso. Nuestro proxy (es decir, P1) intentará establecer una segunda conexión para D2, pero el servidor no la aceptará. Por tanto, la segunda conexión no llegará a establecerse (fallará y D2 se perderá) y solo uno de los bloques de datos será entregado al servidor.
B	El servidor terminaría tras recibir el primer bloque de datos. Falso. La descripción dada en el enunciado no sugiere ese comportamiento. En su lugar, parece que vaya a ser P1 quien aborte al tratar de conectarse por segunda vez.
C	El servidor recibe un solo bloque de datos. Cierto. Es una consecuencia de lo que hemos descrito en el apartado A.
D	El proxy se cierra tras manejar el primer bloque de datos. Falso. No cierra las conexiones ni termina voluntariamente tras haber manejado correctamente el primer bloque de datos.
E	Todas las anteriores.
F	Ninguna de las anteriores.

Considere el fragmento 4 siguiente:

```
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);
```

Asuma que P1 ejecuta este código en vez del fragmento 3, siendo el resto idéntico a lo descrito para el fragmento 3. Conteste las tres cuestiones siguientes:

14. Cuando P2 se conecta al puerto local 8000, P1 …

A	… se conecta a REMOTE_IP. Cierto. La línea 11 implanta ese intento de conexión. Esa línea está ubicada en el *callback* utilizado para manejar conexiones desde otros procesos. P1 está esperando nuevas conexiones en el puerto 8000, como muestra la línea 19.
B	… imprime '`Client connected`' en la consola. Cierto. Eso ocurre en la línea 9, dentro de la función que maneja las conexiones.
C	… crea como máximo una conexión a REMOTE_IP por cada conexión cliente. Cierto. Tal como se ha explicado en el apartado A, esa es la responsabilidad del código mostrado en la línea 11.
D	… espera datos desde REMOTE_IP tan pronto como establezca una conexión con REMOTE_IP. Cierto. Eso se hace en la función que gestiona los eventos "data", mostrada en las líneas 15 a 17. Obsérvese que esa función se ha asociado al objeto "serviceSocket", utilizado para conectar con REMOTE_IP.
E	Todas las anteriores.
F	Ninguna de las anteriores.

15. P2 envía dos bloques grandes de datos en secuencia, D1 seguido por D2, a través de su conexión con P1.
Entonces, en ausencia de fallos …

A	D2 siempre llega a REMOTE_IP tras D1. Sí, pues solo hay una única conexión entre P1 y REMOTE_IP para gestionar todos los mensajes enviados por P2. Esa conexión garantiza orden FIFO y ambos mensajes serán propagados correctamente si no hay fallos.
B	D1 siempre llega a REMOTE_IP tras D2. Falso. Véase la justificación del apartado A.
C	Solo D1 es transmitido a REMOTE_IP. Falso. Véase la justificación del apartado A.
D	Solo D2 es transmitido a REMOTE_IP. Falso. Véase la justificación del apartado A.
E	Todas las anteriores.
F	Ninguna de las anteriores.

16. Asuma que el servidor en REMOTE_IP pueda gestionar solo una conexión a la vez. Entonces, si P2 envía dos bloques de datos …

A	El servidor obtiene ambos bloques de datos. Cierto. Como se ha explicado en la cuestión 15, ambos mensajes se propagan por la misma conexión. P2 gestiona correctamente ambos bloques de datos. Son recibidos sin problemas por el servidor ejecutado en REMOTE_IP.
B	El servidor termina tras recibir el primer bloque de datos. Falso. Véase la justificación del apartado A.
C	El servidor solo recibe un bloque de datos. Falso. Véase la justificación del apartado A.
D	El proxy se cierra tras manejar el primer bloque de datos. Falso. Véase la justificación del apartado A.
E	Todas las anteriores.
F	Ninguna de las anteriores.

17. Ambos fragmentos 3 y 4 …

A	Permiten a P1 cerrar la conexión a REMOTE_IP cuando el cliente cierra la conexión. Falso. No hay ninguna gestión para los eventos "end" (que son los generados cuando una conexión se cierra) en esos dos fragmentos. Por tanto, P1 no puede cerrar la conexión en ese caso.
B	Permiten a P1 cerrar la conexión con sus clientes cuando REMOTE_IP cierre su conexión. Falso. No hay ninguna gestión para los eventos "end" (que son los generados cuando una conexión se cierra) en esos dos fragmentos. Por tanto, P1 no puede cerrar la conexión en ese caso.
C	Necesitan ser modificados para capturar el evento 'end' en `serviceSocket` y `socket` para gestionar adecuadamente el cierre de conexiones. Verdadero. No hay ninguna gestión para los eventos "end" (que son los generados cuando una conexión se cierra) en esos dos fragmentos. Por tanto, debemos extenderlos como sugiere este apartado.
D	No pueden ser modificados para gestionar adecuadamente el cierre de conexiones. Falso. Una gestión adecuada del cierre de conexiones puede implantarse fácilmente añadiendo funciones que actúen como *listeners* de los eventos "end".
E	Todas las anteriores.
F	Ninguna de las anteriores.

Considere el siguiente fragmento 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 fragmento 5, la variable** `WHO`

A	Debe declararse en el ámbito de la función `Remote`.
B	Debe apuntar al objeto que se está construyendo.
C	No puede ser null.
D	No puede ser undefined.
E	Todas las anteriores. El identificador WHO utilizado en ese fragmento debería ser reemplazado por la cadena `this`. Con ello, todos los apartados anteriores son ciertos: está en el ámbito del constructor de `Remote`, representa al objeto que estamos construyendo, no puede ser `null` y no puede ser `undefined`.
F	Ninguna de las anteriores.

19. **En el fragmento 5, `X_WHAT` debe sustituirse por...**

A	El socket de la conexión cliente. Cierto. Debería ser sustituido por la cadena `socket`. Esa variable mantiene el socket con la conexión establecida con el cliente.
B	El socket de la conexión del servidor. Falso. Véase la justificación del primer apartado.
C	El objeto Remote. Falso. Véase la justificación del primer apartado.
D	El objeto servidor. Falso. Véase la justificación del primer apartado.
E	Todas las anteriores.
F	Ninguna de las anteriores.

20. **En el fragmento 5, `X_SERVER_PORT` y `X_SERVER_IP` ...**

A	Se refieren a los atributos `targetHost` y `targetPort` del objeto Remote.
B	Deben sustituirse por `this.targetHost` y `this.targetPort`, respectivamente.
C	Pueden ser idénticos para diferentes objetos Remote.
D	No pueden ser null o undefined
E	Todas las anteriores. Todos los apartados anteriores implican lo mismo: X_SERVER_PORT y X_SERVER_IP corresponden a `this.targetPort` y `this.targetHost`, respectivamente. Diferentes objetos Remote pueden tener los mismos valores en esos atributos. JavaScript no impone ninguna restricción que lo impida. No pueden ser `null` ni `undefined` pues con esos valores no se podría establecer ninguna conexión.
F	Ninguna de las anteriores.