

PRÁCTICA 1

CHATTY

Fèlix Andrés Navarro
Jordi Calatayud Álvarez

Índice

PRÁCTICA 1.....	1
Índice.....	2
Introducción de la práctica.....	3
Client.erl.....	3
Server.erl.....	4
Server2.erl.....	5
Experiment 1.....	7
Testing questions.....	9
Experiment 2.....	10
Robust questions.....	11

Introducción de la práctica

La práctica nos presenta el código de una aplicación que permite a múltiples clientes conectarse a uno o varios servidores, dependiendo de la versión, para comunicarse entre ellos.

Los códigos que se nos da son el correspondiente al que usaría el cliente y dos correspondientes a distintas versiones del servidor. Estos códigos nos vienen con huecos y por tanto tenemos que completarlos nosotros a partir de la información que podemos deducir de los código servidor y cliente.

Client.erl

Este es el código de Client.erl completado:

```
-module(client).
%% Exported Functions
-export([start/2]).

%% API Functions
start(ServerPid, MyName) ->
    ClientPid = spawn(fun() -> init_client(ServerPid, MyName) end),
    process_commands(ServerPid, MyName, ClientPid).

init_client(ServerPid, MyName) ->
    ServerPid ! {client_join_req, MyName, self()},
    process_requests().

%% Local Functions
%% This is the background task logic
process_requests() ->
    receive
        {join, Name} ->
            io:format("[JOIN] ~s joined the chat~n", [Name]),
            %% TODO: ADD SOME CODE
            process_requests();
        {leave, Name} ->
            io:format("[LEAVE] ~s leaved the chat~n", [Name]),
            %% TODO: ADD SOME CODE
            process_requests();
        {message, Name, Text} ->
            io:format("[~s] ~s", [Name, Text]),
            %% TODO: ADD SOME CODE
            process_requests();
```

```

        exit ->
            ok
    end.

%% This is the main task logic
process_commands(ServerPid, MyName, ClientPid) ->
    %% Read from standard input and send to server
    Text = io:get_line("-> "),
    if
        Text == "exit\n" ->
            ServerPid ! {client_leave_req, MyName, ClientPid},
            ok;
        true ->
            ServerPid ! {send, MyName, Text}, %% Enviamos el mensaje
            process_commands(ServerPid, MyName, ClientPid)
    end.

```

Server.erl

Este es el código de Server.erl completado:

```

-module(server).
%% Exported Functions
-export([start/0]).

%% API Functions
start() ->
    ServerPid = spawn(fun() -> process_requests([]) end),
    register(myserver, ServerPid).

process_requests(Clients) ->
    receive
        {client_join_req, Name, From} ->
            NewClients = [From|Clients], %% Añadimos el nuevo pid a la
            lista de clientes
            broadcast(NewClients, {join, Name}),
            process_requests(NewClients); %% Volvemos a llamar a la función
            con la lista de clientes actualizadas
        {client_leave_req, Name, From} ->
            NewClients = lists:delete(From, Clients), %% Utilizamos
            lists:delete para eliminar el pid de la lista de clientes
            broadcast(Clients, {leave, Name}), %% Enviamos a todos los
            clientes restantes quién ha abandonado el chat
            From ! exit,
            process_requests(NewClients); %% Volvemos a llamar a la función
    end.

```

```

con la lista de clientes actualizadas
    {send, Name, Text} ->
        broadcast(Clients, {message, Name, Text}), %% Enviamos a todos
los clientes el mensaje
        process_requests(Clients);
    disconnect ->
        unregister(myserver)
end.

%% Local Functions
broadcast(PeerList, Message) ->
    Fun = fun(Peer) -> Peer ! Message end,
    lists:map(Fun, PeerList).

```

Server2.erl

Este es el código de Server2.erl completado:

```

-module(server2).
%% Exported Functions
-export([start/0, start/1]).

%% API Functions
start() ->
    ServerPid = spawn(fun() -> init_server() end), %%Crea un proceso nuevo
en el que se crea el server
    register(myserver, ServerPid).

start(BootServer) ->
    ServerPid = spawn(fun() -> init_server(BootServer) end), %%Crea un
proceso nuevo en el que se crea el server
    register(myserver, ServerPid).

init_server() ->
    process_requests([], [self()]).

init_server(BootServer) ->
    BootServer ! {server_join_req, self()},
    process_requests([], []).

process_requests(Clients, Servers) ->
    receive
        %% Messages between client and server
        {client_join_req, Name, From} ->
            NewClients = [From|Clients], %% Añadimos el nuevo cliente
            broadcast(Servers, {join, Name}), %% Le decimos a nuestros
servidores que envíen el mensaje

```

```

        process_requests(NewClients, Servers); %% Actualizamos la lista
de clientes
        {client_leave_req, Name, From} ->
            NewClients = lists:delete(From, Clients), %% Eliminamos el
cliente
            broadcast(Servers, {leave, Name}), %% Notificamos a los
servidores de que el cliente
            From ! exit,
            process_requests(NewClients, Servers); %% Actualizamos la lista
de clientes
        {send, Name, Text} ->
            broadcast(Servers, {message, Name, Text}), %% Enviamos a todos
los servidores el nuevo mensaje
            process_requests(Clients, Servers);

%% Messages between servers
disconnect ->
    NewServers = lists:delete(self(), Servers), %% Eliminamos el
servidor de la lista
    broadcast(NewServers, {update_servers, NewServers}), %% Envias
a todos los servidores los cambios en la lista
    unregister(myserver);
    {server_join_req, From} ->
        NewServers = [From|Servers], %% Añadimos el nuevo servidor a la
lista
        broadcast(NewServers, {update_servers, NewServers}), %% Envias
a todos los servidores los cambios en la lista
        process_requests(Clients, NewServers); %% Actualizas la lista
de servidores con el nuevo
        {update_servers, NewServers} ->
            io:format("[SERVER UPDATE] ~w~n", [NewServers]),
            process_requests(Clients, NewServers); %% Actualizas la lista
de servidores

    RelayMessage -> %% Whatever other message is relayed to its clients
        broadcast(Clients, RelayMessage),
        process_requests(Clients, Servers)

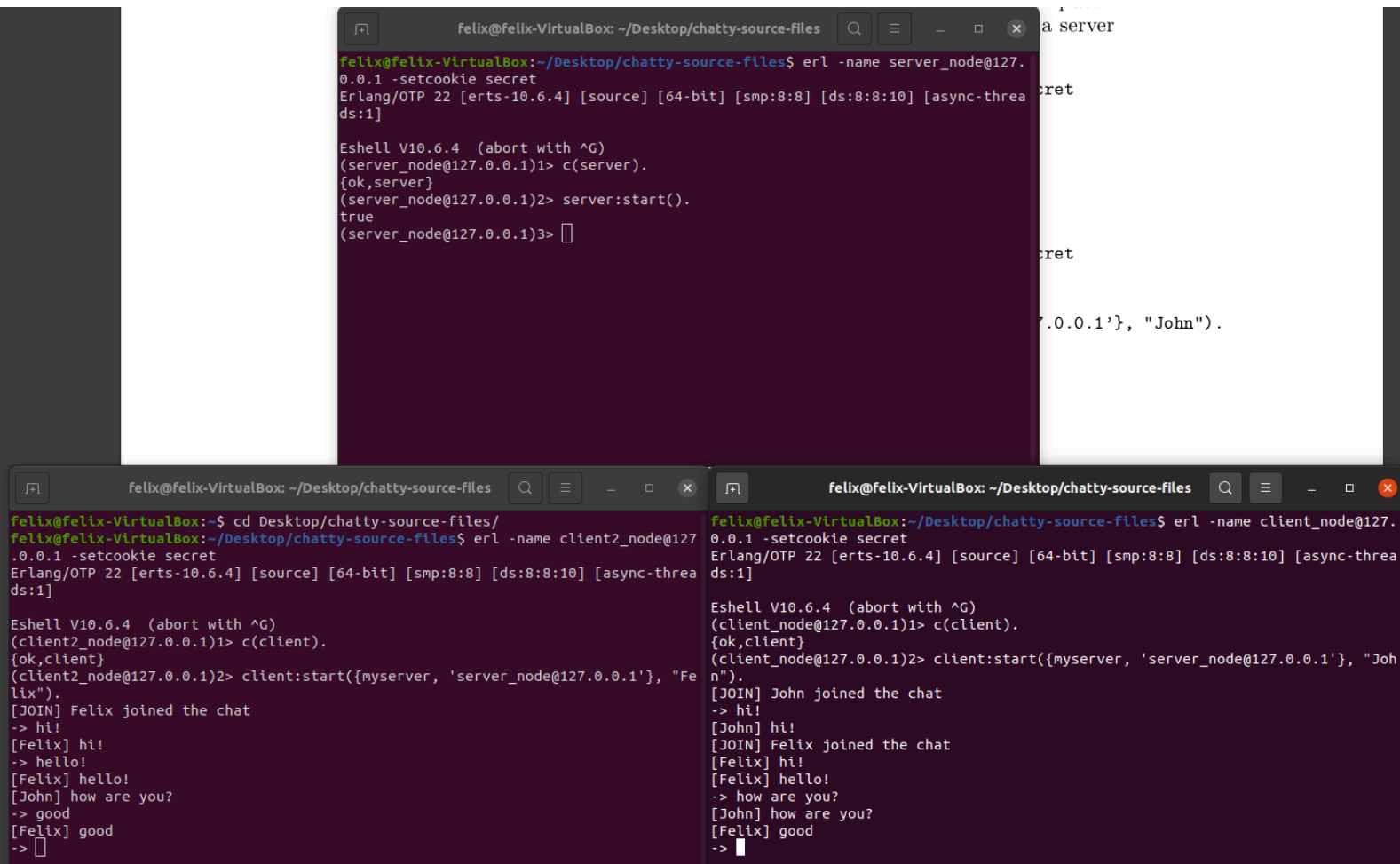
end.

%% Local Functions
broadcast(PeerList, Message) ->
    Fun = fun(Peer) -> Peer ! Message end,
    lists:map(Fun, PeerList).

```

Experiment 1

Try opening 2 or 3 clients and test that all of them receive all the messages. You can also try to communicate among different physical machines (remember to change the local IP by the fully-qualified domain name (or the public IP) of the corresponding machines).



```
felix@felix-VirtualBox: ~/Desktop/chatty-source-files
felix@felix-VirtualBox:~/Desktop/chatty-source-files$ erl -name server_node@127.0.0.1 -setcookie secret
Erlang/OTP 22 [erts-10.6.4] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-thrds:1]

Eshell V10.6.4 (abort with ^G)
(server_node@127.0.0.1)1> c(server).
{ok,server}
(server_node@127.0.0.1)2> server:start().
true
(server_node@127.0.0.1)3>

felix@felix-VirtualBox:~/Desktop/chatty-source-files$ erl -name client2_node@127.0.0.1 -setcookie secret
Erlang/OTP 22 [erts-10.6.4] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-thrds:1]

Eshell V10.6.4 (abort with ^G)
(client2_node@127.0.0.1)1> c(client).
{ok,client}
(client2_node@127.0.0.1)2> client:start([myserver, 'server_node@127.0.0.1'], "Felix").
[JOIN] Felix joined the chat
-> hi!
[Felix] hi!
-> hello!
[Felix] hello!
[John] how are you?
-> good
[Felix] good
->

felix@felix-VirtualBox:~/Desktop/chatty-source-files$ erl -name client_node@127.0.0.1 -setcookie secret
Erlang/OTP 22 [erts-10.6.4] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-thrds:1]

Eshell V10.6.4 (abort with ^G)
(client_node@127.0.0.1)1> c(client).
{ok,client}
(client_node@127.0.0.1)2> client:start([myserver, 'server_node@127.0.0.1'], "John").
[JOIN] John joined the chat
-> hi!
[John] hi!
[JOIN] Felix joined the chat
[Felix] hi!
[Felix] hello!
-> how are you?
[John] how are you?
[Felix] good
->
```

Aquí podem veure com hem iniciat un servidor a una terminal, i hem iniciat 2 clients, cadascun a una terminal different. Aquest experiment l'hem fet al mateix dispositiu (PC).

Després vam provar de fer el mateix pero a dues màquines diferents:

```

felix@felix-VirtualBox: ~/Desktop/chatty-source-files$ erl -name server_node@192.168.1.152 -setcookie secret
Erlang/OTP 24 [erts-12.2.1] [source] [64-bit] [smp:1:1] [ds:1:1:10] [async-thrds:1] [jit]

Eshell V12.2.1 (abort with ^G)
(server_node@192.168.1.152)1> c(server).
{ok,server}
(server_node@192.168.1.152)2> server:start().
true
(server_node@192.168.1.152)3> nodes().
['client_node@192.168.1.221']
(server_node@192.168.1.152)4> 

```

```

felix@felix-VirtualBox: ~/Desktop/chatty-source-files
inet 192.168.1.221/24 brd 192.168.1.255 scope global dynamic noprefixroute enp0s3
    valid_lft 3317sec preferred_lft 3317sec
inet6 fe80::a3aa:61d6:197:5173/64 scope link noprefixroute
    valid_lft forever preferred_lft forever
felix@felix-VirtualBox: ~/Desktop/chatty-source-files$ erl -name client_node@192.168.1.221 -setcookie secret
Erlang/OTP 22 [erts-10.6.4] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-thrds:1]

Eshell V10.6.4 (abort with ^G)
(client_node@192.168.1.221)1> c(client).
{ok,client}
(client_node@192.168.1.221)2> client:start({myserver, 'server_node@192.168.1.152'}, "Felix").
(client_node@192.168.1.221)2> client:start({myserver, 'client_node@192.168.1.152'}, "Felix").
* 2: syntax error before: Felix
(client_node@192.168.1.221)2> client:start({myserver, 'server_node@192.168.1.152'}, "Felix").
[JOIN] Felix joined the chat
-> hi
[Felix] hi
-> 

```

Ens connectem com a clients al servidor, utilitzant la direcció IP de l'altra màquina (192.168.1.152). A l'imatge on iniciem el server a l'altre màquina, es pot apreciar com al fer "nodes()." ens indica que tenim un client 192.168.1.221, que es la IP del client. D'aquesta forma, ja hem fet l'experiment tant a la mateixa màquina, com a màquines diferents.

Testing questions

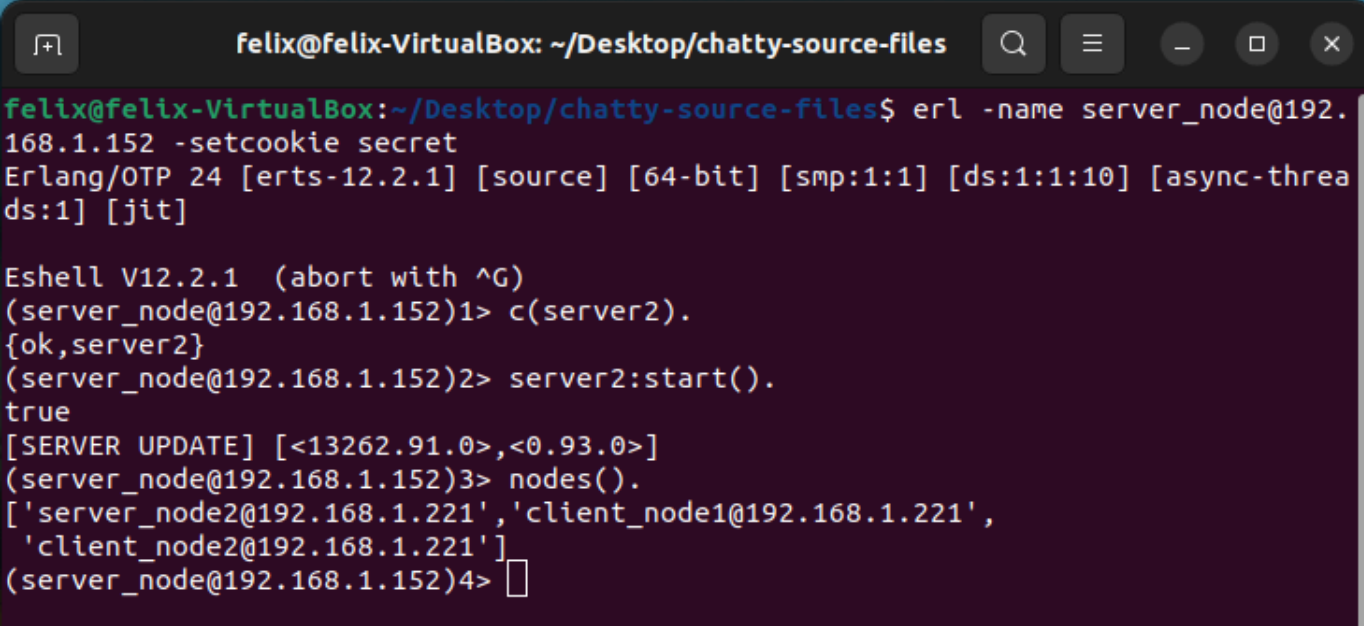
1. Does this solution scale when the number of users increase?
No, puesto que utilizamos el mismo servidor independientemente del número de usuarios.
2. What happens if the server fails?
Si el servidor falla, no aparecerán los mensajes que se envíen y se deberá de ejecutar todo, tanto cliente como servidor, para que la aplicación funcione otra vez.
3. Are the messages from a single client guaranteed to be delivered to any other client in the order they were issued?
Sí, porque primero se envía y después vuelve a salir la opción para enviar un mensaje de nuevo.
4. Are the messages sent concurrently by several clients guaranteed to be delivered to any other client in the order they were issued?
No se asegura que el orden de los mensajes sea consistente cuando varios clientes están enviando mensajes simultáneamente.
5. Is it possible that a client receives a response to a message from another client before receiving the original message from a third client?
Sí, es posible que un cliente reciba una respuesta de otro cliente antes de recibir el mensaje original de un tercer cliente. Esto ocurre porque en Erlang los mensajes de diferentes procesos llegan de manera asíncrona y no hay garantía de orden entre ellos. El servidor simplemente retransmite los mensajes a los clientes, y estos los procesan en el orden en que los reciben, que puede variar debido a la concurrencia del sistema.
6. If a user joins or leaves the chat while the server is broadcasting a message, will he/she receive that message?
No, un usuario que se une o abandona el chat mientras el servidor está transmitiendo un mensaje no recibirá ese mensaje. Esto se debe a que el servidor usa la lista de clientes existente al momento de iniciar la transmisión. Los usuarios que se unan después de que el mensaje ya esté en proceso de envío no estarán en esa lista, y los que se desconecten en medio de la transmisión ya no recibirán mensajes.

Experiment 2

Once you have a set of servers up and running, try connecting some clients to each of the server instances and begin to chat. Does it work?

You can also try to crash some of the servers and see what happens.

Aquí podem veure com iniciem un server (server2) a una terminal, i com tenim 2 clients i un altre servidor connectats a nosaltres.



```
felix@felix-VirtualBox: ~/Desktop/chatty-source-files
felix@felix-VirtualBox:~/Desktop/chatty-source-files$ erl -name server_node@192.168.1.152 -setcookie secret
Erlang/OTP 24 [erts-12.2.1] [source] [64-bit] [smp:1:1] [ds:1:1:10] [async-threa
ds:1] [jit]

Eshell V12.2.1 (abort with ^G)
(server_node@192.168.1.152)1> c(server2).
{ok,server2}
(server_node@192.168.1.152)2> server2:start().
true
[SERVER UPDATE] [<13262.91.0>,<0.93.0>]
(server_node@192.168.1.152)3> nodes().
['server_node2@192.168.1.221','client_node1@192.168.1.221',
'client_node2@192.168.1.221']
(server_node@192.168.1.152)4> █
```

En aquesta imatge tenim un servidor connectat al servidor que hem iniciat abans.



```
felix@felix-VirtualBox:~/Desktop/chatty-source-files$ erl -name server_node2@192.168.1.221 -setcookie secret
Erlang/OTP 22 [erts-10.6.4] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threa
ds:1]

Eshell V10.6.4 (abort with ^G)
(server_node2@192.168.1.221)1> c(server2).
{ok,server2}
(server_node2@192.168.1.221)2> server2:start({myserver, 'server_node@192.168.1.152'}).
true
[SERVER UPDATE] [<0.91.0>,<12472.93.0>]
(server_node2@192.168.1.221)3> █
```

```
felix@felix-VirtualBox: ~/Desktop/chatty-source-files
Eshell V10.6.4 (abort with ^G)
(client_node2@192.168.1.221)1> c(client).
{ok,client}
(client_node2@192.168.1.221)2> client:start([myserver, 'server_node@192.168.1.221'], "Jordi").
->
BREAK: (a)bort (c)ontinue (p)roc info (l)nfo (l)oaded
(v)ersion (k)ill (D)b-tables (d)istribution
^CFelix@felix-VirtualBox:~/Desktop/chatty-source-files$ erl -name client_node2@192.168.1.221 -setcookie secret
Erlang/OTP 22 [erts-10.6.4] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threa
ds:1]

Eshell V10.6.4 (abort with ^G)
(client_node2@192.168.1.221)1> c(client).
{ok,client}
(client_node2@192.168.1.221)2> client:start([myserver, 'server_node@192.168.1.152'], "Jordi").
[JOIN] Jordi joined the chat
-> hello
[Jordi] hello
-> hi
[Jordi] hi
->

felix@felix-VirtualBox:~/Desktop/chatty-source-files$ cd Desktop/chatty-source-files/
felix@felix-VirtualBox:~/Desktop/chatty-source-files$ erl -name client_node1@192.168.1.221 -setcookie secret
Erlang/OTP 22 [erts-10.6.4] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threa
ds:1]

Eshell V10.6.4 (abort with ^G)
(client_node1@192.168.1.221)1> c(client).
{ok,client}
(client_node1@192.168.1.221)2> client:start([myserver, 'server_node@192.168.1.152'], "Felix").
[JOIN] Felix joined the chat
[JOIN] Jordi joined the chat
[Jordi] hello
[Jordi] hi
->
```

I per últim, tenim 2 clients, connectats a un servidor diferent cadascun.

Veiem que funciona perfectament, i al tancar un dels servidors, veurem com el client que estava connectat a aquest servidor perdrà la comunicació, però l'altre no.

Robust questions

1. What happens if a server fails?
Si un servidor se desconecta, lo que ocurra dependerá de si hay más servidores creados o no. Si solo hay un servidor creado, al desconectarse este, la aplicación dejará de funcionar al igual que pasaba en el antiguo servidor. Si hay otro servidor aparte del eliminado, la aplicación funcionará correctamente.
2. Do your answers to previous questions iii, iv, and v still hold in this implementation?
Las respuestas de dichas preguntas se mantiene puesto que hacen referencia al manejo de mensajes por parte del servidor y el cliente y en esta nueva versión del servidor ese apartado no se toca, sino que se mejora la escalabilidad y la persistencia de la aplicación, con la posibilidad de tener varios servidores que reparten sus tareas.
3. What happens if there are concurrent requests from servers to join or leave the system?
Esto crearía una condición de carrera (Race Condition), ya que los servidores podrían actualizar la lista de manera inconsistente. Cada servidor podría comunicar a los demás una lista parcial de servidores, lo que causaría una desincronización entre los servidores.

4. What are the advantages and disadvantages of this implementation regarding the previous one?

Ventajas:

Si un servidor se desconecta, solo algunos clientes pierden la conexión, mientras que el resto del sistema sigue funcionando. La carga de trabajo se distribuye entre varios servidores, lo que favorece la escalabilidad.

Desventajas:

El sistema es más complicado de gestionar y mantener debido a su mayor complejidad. Podrían surgir problemas al intentar conectar o desconectar varios servidores al mismo tiempo, aunque es algo que puede resolverse fácilmente mediante el uso de un servidor centralizado que inicie todos los servidores.