Patrons

Reactor

Patró Reactor

El patró reactor és conegut també com dispatcher o Notifier.

S'usa per una aplicació que:

- gestiona esdeveniments
- ha de reaccionar a diverses peticions quasi simultàniament, però les processa síncronament i en l'ordre d'arribada.

Exemples

- servidors amb múltiples clients
- Interfícies d'usuari amb diverses fonts d'esdeveniments
- serveis de transaccions
- centraleta

Comportament exigit:

- l'aplicació no ha de bloquejar innecessàriament altres peticions mentre s'està gestionant una petició
- ha de ser fàcil incorporar nous tipus d'esdeveniments i peticions
- la sincronització ha d'estar amagada per facilitar la implementació de l'aplicació

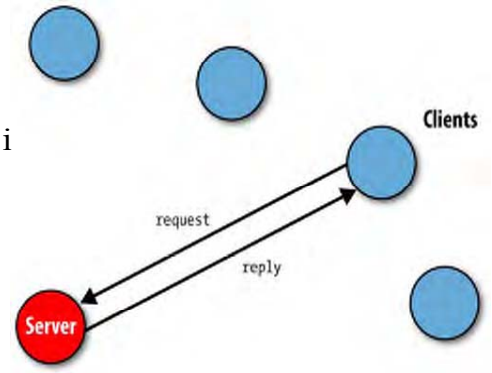
Exemple: Models Client / Servidor

- Els clients i servidors es poden representen com a processos d'Erlang.

- Un servidor pot ser:
  - Una cua FIFO per una impressora,
  - Un gestor de finestres,
  - Un servidor de fitxer.
- Els recursos que utilitza podria ser:
  - Una base de dades,
  - Un calendari
  - Una llista finita d'elements (habitacions, llibres, o freqüències de ràdio)

- Els clients utilitzen aquests recursos mitjançant l'enviament de les peticions (missatges) al servidor per:
  - Imprimir un arxiu,
  - Actualitzar una finestra,
  - Reservar una habitació, un llibre o utilitzar una freqüència.

- El servidor rep la petició (el missatge), l'avalua, i respon:
  - amb un reconeixement i un valor si la petició s'ha realitzat correctament,
  - o amb un error si la petició no ha tingut èxit.



Exemple: Models Client / Servidor

- Comportament exigit:

- l'aplicació no ha de bloquejar innecessàriament altres peticions mentre s'està gestionant una petició:
  - El servidor espera en un bucle central els esdeveniments que vagin arribant.
  - Un cop s'ha rebut un esdeveniment es trasllada el seu processament a un gestor específic (procés)

- Ha de ser fàcil incorporar nous tipus d'esdeveniments i peticions:
  - El reactor permet afegir / treure gestors dels esdeveniments.

- La sincronització ha d'estar amagada per facilitar la implementació de l'aplicació:
  - El pas de missatges sovint s'oculta en les interfícies funcionals, així que en lloc d'enviar el missatge:

```
printerserver!{print, File}
```

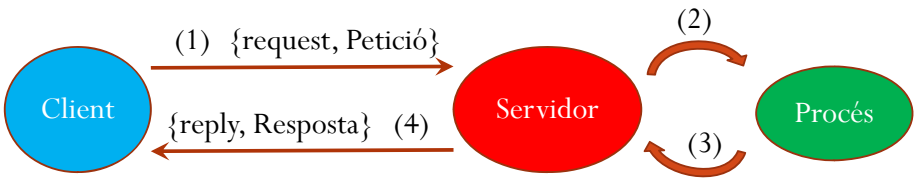
un client farà la crida:

```
printerserver:print(File)
```

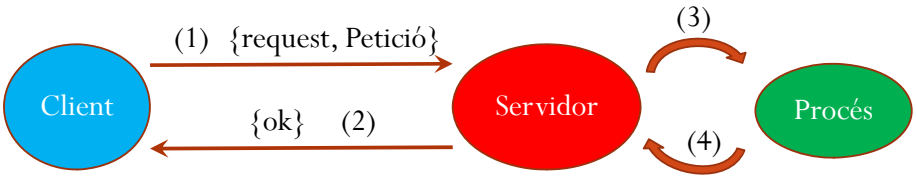
Aquesta és una manera d'amagar informació, en la que el client no és conscient de que el servidor és un procés, que podria estar registrat, i que podria residir en un equip remot. Tampoc es mostra el protocol de missatges que s'utilitza entre el client i el servidor, mantenint el contacte entre ells segur i simple.
El client necessita fer una crida a una funció i esperar un valor de retorn.

Exemple: Models Client / Servidor

- **Comportament exigit:**
- Si un client que utilitza un servei o un recurs del servidor espera una resposta a la petició, la crida al servidor serà síncrona.
- Les crides síncrones retornen el valor esperat pel client. Els retorns de valors sol seguir el format:
  - {ok, Resultat}
  - {error, Motiu}

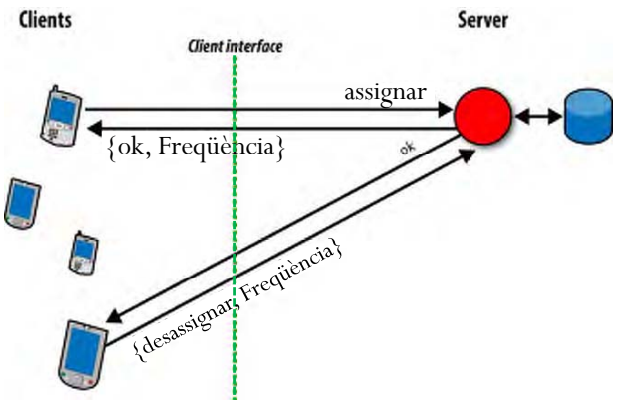


- Si el client no necessita una resposta, la crida al servidor pot ser asíncrona. Les crides asíncrones normalment tornen l'àtom de correcte (ok), el que indica que la petició s'està atenent al servidor.

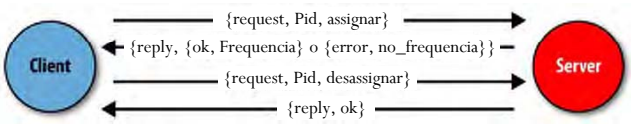


Exemple: Models Client / Servidor. Descripció del problema

- Es vol gestionar l'assignació de freqüències de ràdio per uns telèfons mòbils connectats a la xarxa. El telèfon demana que se li assigni una freqüència cada vegada que fa una trucada, i l'allibera quan la trucada ha acabat.
- Els client seran els diferents telèfons que poden intentar establir una trucada
- El servidor serà el que assigna les freqüències de ràdio
- Quan un telèfon mòbil ha establert una connexió amb un altre abonat, crida a la funció del client per demanar una freqüència *freqüència:assignar()*. Aquesta funció genera un missatge síncron (assignar) que s'envia al servidor.
- El servidor controla i respon amb un missatge que conté una freqüència disponible {ok, Freqüència} o un error si totes les freqüències estan utilitzades {error,no_freqüències }.



- Quan el client finalitza la trucada telefònica ha d'alliberar la connexió, s'ha de cancel·lar l'assignació de la freqüència perquè altres clients la pugin utilitzar. I ho fa cridant a la funció de client freqüència:desassignar (Freqüència). La crida genera un missatge {desassignar, Freqüència} que s'envia al servidor.
- El servidor pot assignar la freqüència disponible a altres clients i respon amb l'àtom de correcte (ok).



Exemple: Models Client / Servidor: Solució

- La primera part del codi del mòdul d'assignació de les freqüències, corresponent al servidor, pot ser:

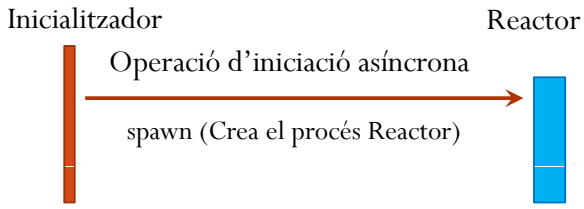
```
-module(frequencia).
-export([iniciar/0, stop/0, assignar/0, desassignar/1]).
-export([init/0]).

% Funcions d'inicialització per crear i inicialitzar el servidor.
iniciar() ->
    register(frequencia, spawn(frequencia, init, [])).

init() ->
    Frequencies = {get_frequencies(), []},
    loop(Frequencies).

% Codificat
get_frequencies() -> [10,11,12,13,14,15].
```

La funció *iniciar/0* genera un procés init/0 que inicia l'execució del mòdul de freqüència.
spawn retorna el pid del procés que es passa com a segon paràmetre a la funció (¡BIF) *register*. El primer paràmetre és l'àtom *frequencia*, que és l'alies amb el que estem registrant el procés.
Aquesta assignació segueix el conveni de registrar un procés amb el mateix nom que el mòdul en què es defineix.
El nou procés (*init*) crea una tupla amb la llista de les freqüències disponibles (funció: *get_frequencies/0)*, i una llista de les freqüència assignades ([]). La tupla, serà el que anomenem les dades o cicle d'estat, s'afegeix a la variable *Frequencies,* que s'usa com a paràmetre de la funció de recepció avaluació (reactor), que en aquest exemple s'ha anomenat loop/1.



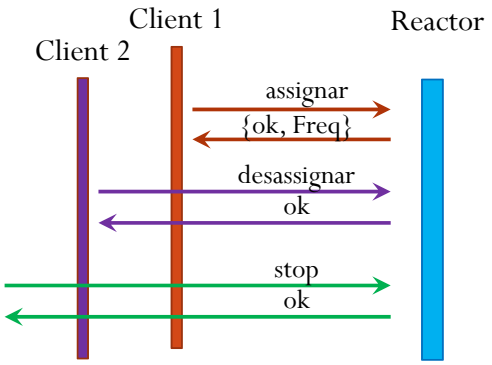
Exemple: Models Client / Servidor: Solució

```
%  Les funcions del client
stop()      -> call(stop).
assignar() -> call(assignar).
desassignar(Freq) -> call({desassignar, Freq}).

% Amaguem el pas de missatges i el protocol de missatge en una interfície funcional.
call(Missatge) ->
    frequencia ! {request, self(), Missatge},
    receive
        {reply, Reply} -> Reply
    end.
```

- Funcions del client

Al tractar-se d'una transferència síncrona, quan s'ha enviat el missatge, el client es queda esperant la resposta
La resposta del servidor {reply, Reply} és l'àtom 'reply', i la variable *Reply* retorna el valor de les funcions de client.



La crida a la funció *call/1*, passa el missatge que s'ha d'enviar al servidor com un argument. Aquesta funció encapsula el protocol de missatges entre el servidor i els seus clients, enviant un missatge amb el format {request, Pid, Missatge}. L'àtom *request* és una etiqueta a la tupla, Pid és l'identificador del procés que el crida (retorn de la funció BIF self()), i *Missatge* és el missatge que es vol enviar.

Exemple: Models Client / Servidor: Solució

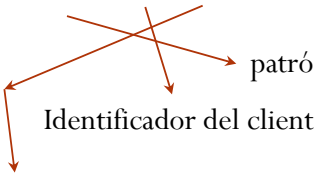
```
% El bucle principal
loop(Frequencies) ->
    receive
        {request, Pid, assignar} ->
            {NovesFrequencies, Reply} = assignar(Frequencies, Pid),
            reply(Pid, Reply),
            loop(NovesFrequencies);
        {request, Pid , {desassignar, Freq}} ->
            NovesFrequencies = desassignar(Frequencies, Freq),
            reply(Pid, ok),
            loop(NovesFrequencies);
        {request, Pid, stop} ->
            reply(Pid, ok)
    end.

reply(Pid, Reply) ->
    Pid ! {reply, Reply}.
```

- Funcions del servidor: loop/1

Missatges:

{request, Pid, Missatge}



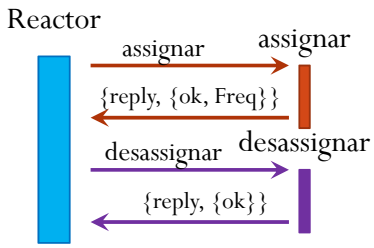
Acceptarà rebre tres tipus de peticions:

- assignar,
- desassignar i
- stop.

El *Missatge* és el patró lligat a l'expressió i s'utilitza per determinar quin clàusula s'ha d'executar.

Aquestes funcions retornen un respostes al client, que en aquest cas consisteix en la freqüència assignada o un ok.

El *pid* del client s'ha enviat com a part de la petició, i s'utilitza per identificar el procés que l'ha cridat i retornar-li la resposta a *reply/2*.



Exemple: Models Client / Servidor; Solució

- Un client vol iniciar una trucada i crida a la funció *frequencia:assignar()*.
- Aquesta funció enviarà un missatge amb el format {request, Pid, assignar} al servidor de freqüències. Aquest missatge activarà la funció del servidor *assignar(Frequencies, Pid)*, on *Frequencies* és la tupla de les freqüències assignades i disponibles. La funció *assignar* comprovarà si hi ha freqüències disponibles:
  - Si hi ha freqüències disponibles, retorna la variable "NovesFrequencies"actualitzada, on la freqüència assignada s'ha mogut de la llista de freqüències disponible a la llista de freqüències assignades juntament amb el pid del client. La resposta (freqüència assignada) s'envia al client de la forma {*ok, Frequencia assinada*}.
  - Si no hi ha freqüències disponibles, les dades del bucle "NovesFrequencies"es torna sense canvis i s'envia el missatge {*error, no_frequencia* } com a resposta al client.
- La resposta al client s'envia amb la funció *reply(Pid, Missatge)*, que dóna format al missatge intern client / servidor i l'envia de tornada al client.
- La crida a la funció *bucle/1* es fa de forma recursiva, i es passen les noves dades del bucle com paràmetres.
- La funció *desassignar* funciona d'una manera similar. La funció client envia un missatge amb el format {request, Pid, desassignar} que es correspon amb el segon paràgraf de la declaració rebre del servidor. Fa una crida a la funció *desassignar(Frequencies, Freq)* i elimina la freqüència de la llista d'assignats a la de disponibles, i torna les dades actualitzades al bucle. L'àtom correcte (ok) s'envia de tornada al client, i es crida la funció *bucle/1* recursivament amb les dades actualitzades.
- Si es rep la petició d'aturada 'stop', es retorna 'ok' al procés que l'ha cridat i el servidor finalitza l'execució, ja que no hi ha cap més crida recursiva.

Exemple: Models Client / Servidor: Solució

- Funcions del servidor

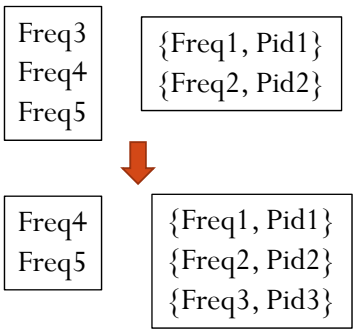
```
% Funcions internes per assignar i alliberar freqüències.

assignar({[], Assignat}, _Pid) ->
{{[], Assignat}, {error, no_frequencia}};

assignar({[Freq|Lliure], Assignat}, Pid) ->
{{Lliure, [{Freq, Pid}|Assignat]}, {ok, Freq}}.

desassignar({Lliure, Assignat}, Freq) ->
NouAssignat=lists:keydelete(Freq, 1, Assignat),
      {[Freq|Lliure],  NouAssignat}.
```

{[Freq | Lliure], Assignat}



assignar/2

- Si no hi ha freqüències disponibles, *assignar/2* coincideix amb el patró de la primera clàusula, ja que el primer element de la tupla que conté la llista de freqüències disponibles està buida. Retorna la tupla {error, no_frequencia} juntament amb de les dades del bucle sense canvis.
- Si com a mínim hi ha una freqüència disponible coincidirà amb la segona clàusula. La freqüència s'elimina de la llista de les freqüències disponibles, i es passa a la llista de les freqüències assignades juntament amb el pid del client i.

desassignar/2

- Eliminà de la llista de freqüències assignades la freqüència *Freq* utilitzant la funció de la llibreria *lists:keydelete/3* i l'afegeix a la llista de freqüències disponibles.

Retorna una llista de tuples que s'ha eliminat la tupla que conté l'element *key* de la posició de la tupla *N*

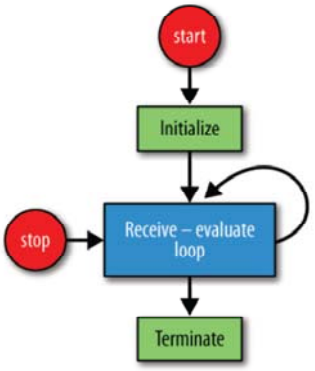
Exemple: Models Client / Servidor: Solució

- simulació

```
1> frequencia:iniciar().
true
2> frequencia:assignar().
{ok,10}
3> frequencia:assignar().
{ok,11}
4> frequencia:assignar().
{ok,12}
5> frequencia:assignar().
{ok,13}
6> frequencia:desassignar(11).
ok
7> frequencia:assignar().
{ok,11}
8> frequencia:stop().
ok
```

Exemple d'un Patró de processos

- L'esquelet del procés d'una aplicació, ja sigui un navegador web o un processador de textos, que s'ocupa de moltes finestres obertes simultàniament controlades per un gestor central de finestres.
- L'objectiu és tenir un procés per a cada activitat concurrent, el camí a seguir, es generar un procés per a cada finestra. Processos no registrats, ja que podria haver-hi moltes finestres del mateix tipus funcionant al mateix temps.
- Cada procés crida a la funció *initialize*, que dibuixa i mostra la finestra i el seu contingut. El retorn de la funció conté les referències als controls que es mostren a la finestra. Aquestes referències s'emmagatzema en la variable d'estat i s'utilitzen sempre que la finestra s'hagi d'actualitzar. La variable d'estat s'usa com a paràmetre o argument de la funció recursiva que implementa el bucle de recepció i avaluació.
- La funció *loop*, espera que arribin les peticions (esdeveniments) de les finestres. Podria ser un usuari escrivint en un formulari o triar una opció del menú, o un procés extern dones dades que s'han d'anar mostrant. Cada esdeveniment relacionat amb aquesta finestra es tradueix a un missatge d'Erlang que s'envia al procés.



Exemple d'un Patró de processos

- El procés, en rebre el missatge, crida a la funció *handle_msg*, passant-li el missatge i l'estat com a paràmetres. Si l'esdeveniment s'ha generat:
  - degut a que s'han premut unes tecles del teclat, la funció *handle* pot ser que hagi de mostrar-les.
  - si l'usuari ha escollit una entrada en un dels menús, la funció *handle* ha de prendre les mesures adequades en l'execució d'aquesta opció del menú.
  - si el succés be d'un procés extern de dades, com la imatge d'una càmera web o d'un missatge d'alerta, s'ha d'utilitzar el control adequat.
- La recepció d'aquests esdeveniments en Erlang és vist com un model genèric en tots els processos. Què es considera específic i canviar d'un procés a un altre depèn de com es gestionin aquests esdeveniments.
- Quan el procés rep un missatge de detenció (stop). Aquest missatge es pot haver originat:
  - perquè un usuari ha escollit l'opció de menú Sortir,
  - fen clic al botó 'destruir',
  - des del gestor de finestres transmetent una notificació de que l'aplicació es tancarà.
- Independentment de l'origen, s'envia al procés un missatge de detenció. Al rebre'l, el procés crida la funció de finalització, que destrueix tots els dispositius, assegurant que ja no es mostren i el procés finalitza l'execució.

Exemple d'un Patró de processos

```
-module(server).
-export([start/2, stop/1, call/2]).
-export([init/1]).

start(Name, Data) ->
    Pid = spawn(generic_handler, init, [Data]),
    register(Name, Pid),
    ok.

stop(Name) ->
    Name ! {stop, self()},
    receive {reply, Reply} ->
        Reply
    end.

call(Name, Msg) ->
    Name ! {request, self(), Msg},
    receive {reply, Reply} ->
        Reply
    end.
```

Exemple d'un Patró de processos

```
reply(To, Msg) ->
    To ! {reply, Msg}.

init(Data) ->
    loop(initialize(Data)).

loop(State) ->
    receive
        {request, From, Msg} ->
            {Reply,NewState} = handle_msg(Msg,State),
            reply(From, Reply),
            loop(NewState);
        {stop, From}  ->
            reply(From, terminate(State))
    end.

initialize(...)   -> ...

handle_msg(...,...) -> ...

terminate(...)    -> ...
```