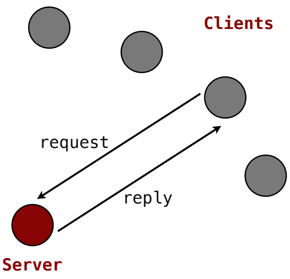
Erlang Solutions Ltd.

Process Design Patterns

Process Design Patterns

- Client Server Models
- A Server Example
- Finite State Machines
- Event Handlers

Client Server Models



Clients

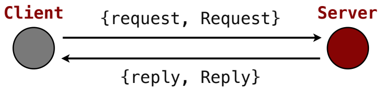
request

reply

Server

- Processes can be used to implement client server solutions
- A server is usually responsible for providing a service or handling a resource
- Clients are the processes which use these resources

Client Server Models

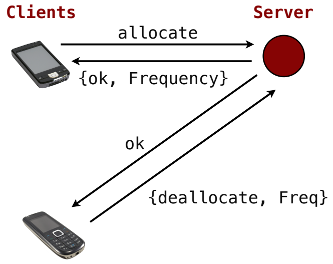


Client {request, Request} Server

{reply, Reply}

- Clients make requests to the server through message passing
- Message passing is often hidden in functional interfaces
- If the client using the service needs a reply to the request, the call to the server has to be **synchronous**
- If the client does not need a reply, the call to the server can be **asynchronous**

A Server Example



Clients

allocate

Server

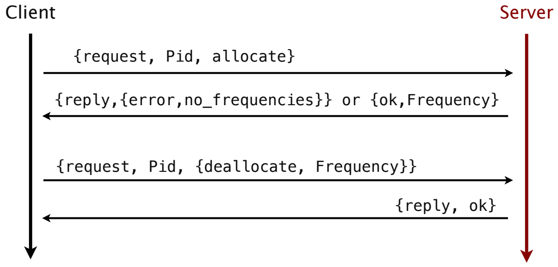
{ok, Frequency}

ok

{deallocate, Freq}

- The following server is responsible for allocating and de-allocating frequencies on behalf of mobile phones

A Server Example



Client

Server

{request, Pid, allocate}

{reply,{error,no_frequencies}} or {ok,Frequency}

{request, Pid, {deallocate, Frequency}}

{reply, ok}

A Server Example

```
-module(frequency).
-export([start/0, stop/0,  allocate/0, deallocate/1]).
-export([init/0]).

start() ->
    register(frequency, spawn(frequency, init, [])).

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

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

A Server Example

```
stop()           -> call(stop).
allocate()       -> call(allocate).
deallocate(Freq) -> call({deallocate, Freq}).

%% We hide all message passing and the message protocol in
%% functional interfaces.
call(Message) ->
    frequency ! {request, self(), Message},
    receive
        {reply, Reply} -> Reply
    end.

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

A Server Example

```
loop(Frequencies) ->
  receive
    {request, Pid, allocate} ->
       {NewFrequencies,Reply} = allocate(Frequencies,Pid),
       reply(Pid, Reply),
       loop(NewFrequencies);
    {request, Pid , {deallocate, Freq}} ->
       NewFrequencies = deallocate(Frequencies, Freq),
       reply(Pid, ok),
       loop(NewFrequencies);
    {request, Pid, stop} ->
       reply(Pid, ok)
    end.
```

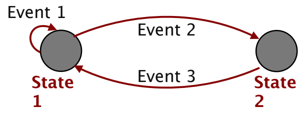
A Server Example

```
%% The Internal Functions
%% Help functions used to allocate and deallocate
frequencies.

allocate({[], Allocated}, Pid) ->
    {{[], Allocated}, {error, no_frequency}};
allocate({[Freq|Free], Allocated}, Pid) ->
    {{Free, [{Freq, Pid}|Allocated]}, {ok, Freq}}.

deallocate({Free, Allocated}, Freq) ->
    NewAllocated = lists:keydelete(Freq, 1, Allocated),
    {[Freq|Free],  NewAllocated}.
```

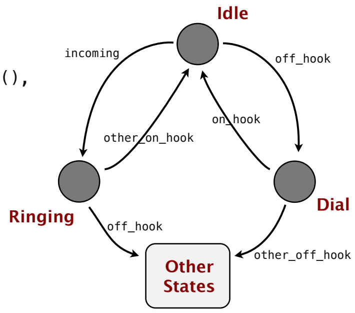
Finite State Machines



- Processes can be used to implement finite state machines
- Each state is represented as a tail recursive function
- Each event is represented as an incoming message
- Each state transition is achieved by calling the function denoting the new state

Finite State Machines: example

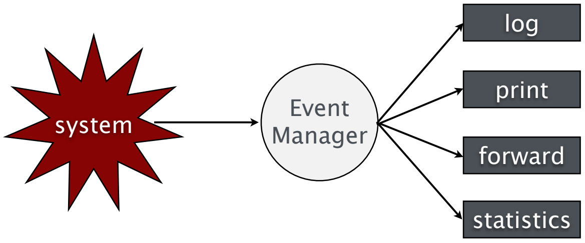
```
idle() ->
  receive
     {A, incoming} ->
           start_ringing(),
         ringing(A);
     {A, off_hook} ->
         start_tone(),
           dial(A)
   end.
```



Event Handlers

- Processes can be used to implement event handlers
- A handler will receive a specific type of event
  - Alarms
  - Equipment state changes
  - Errors
- When an event is received, one or more functions are applied on the event
- Some or all of these actions can be enabled and disabled during run time

Event Handlers: example



- Alarm managers are implemented as event handlers

Process Design Patterns

- Client Server Models
- A Server Example
- Finite State Machines
- Event Handlers