Chapter 12

More Erlang

Recursion

Erlang optimizes all recursion so tail recursion doesn't necessarily improve efficiency

Factorial with guard

```
f(N) when N > 0 -> N * f(N - 1);
f(0) -> 1.
```

The Scheduler

Allocating execution time among processes:

- Every command is assigned a number of *reduction* steps
- This value is reduced for every operation executed
- If a process's reduction count reaches zero, it is *preempted*

`receive`

Similar to a `case` statement, but the process is *suspended* in a `receive` statement until a message is matched

At any given time, most processes will be suspended in `receive` statements waiting for events to trigger actions

`receive after`

Allows a process to continue after a given delay even if no messages are received

Concurrency-Oriented Programming

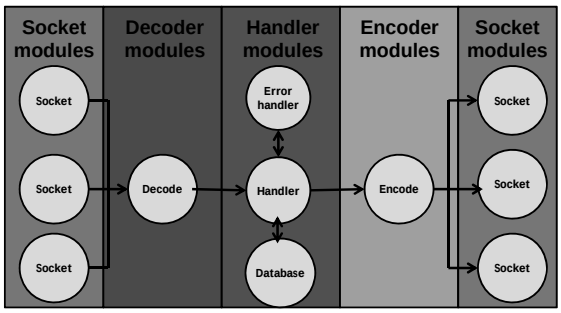
Create a process for every truly concurrent *activity* in a system

- Not for every *task*

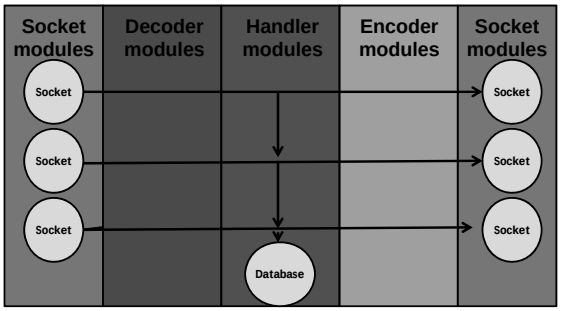
Case Study – IM Proxy

- System receives a packet through a socket
- Decodes it
- Takes actions based on its content
- Encodes a reply packet
- Sends reply to a different socket

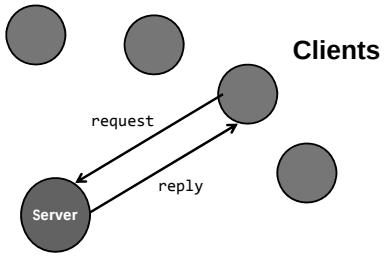
Poor Design – A process for every task



Good Design – A process for each concurrent activity



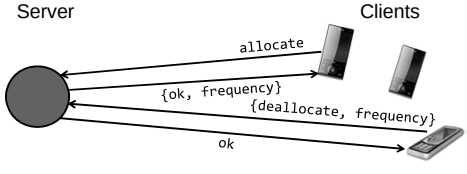
Client/server Models



Client/Server Example

Server manages radio frequencies for cell phones connected to the network

Phone requests a frequency when a call needs to be connected and releases it when call is terminated



Server Functions

```
%% The start function creates and initializes
%% the server.

start() ->
   register(frequency, spawn(frequency, init, [])).
init() ->
  Frequencies = {get_frequencies(), []},
  loop(Frequencies).

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

Client Functions

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

%% Functional interface to message passing
call(Message) ->
   frequency ! {request, self(), Message},
   receive
      {reply, Reply} -> Reply
   end.
```

Main Receive-Evaluate Loop

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

Erlang Term Storage (ETS)

Used for *collections* of items with efficient storage and retrieval

ETS tables store tuples with access using a *key field*

Set: No duplicates

Ordered set: Can be traversed in lexicographical order based on keys

Bag: Allows duplicate entries for the same key

Duplicate bag: Allows duplicate elements (key & value)

All use hash table implementation except *Ordered set* which uses AVL balanced binary tree

Credits

Erlang Programming, by Francesco Cesarini and Simon Thompson